Sun Performance Library Reference Manual

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-2461-10
May 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Sun Performance Library(TM) Reference Manual

Sun Forte(TM) Developer 7

This reference manual is a PDF version of the section 3P man pages. For additional information, see the *Sun Performance Library User's Guide*, available on `docs.sun.com`, or the *LAPACK Users' Guide*, available from the Society for Industrial and Applied Mathematics (SIAM).

---

ccsrmm - ccsrmm - compressed sparse row format matrix-matrix multiply

ccsrsm - ccsrsm - compressed sparse row format triangular solve

cdiamm - cdiamm - diagonal format matrix-matrix multiply

cdiasm - cdiasm - diagonal format triangular solve

cdotc - cdotc - compute the dot product of two vectors conjg(x) and y.

cdotci - cdotci - Compute the complex conjugated indexed dot product.

cdotu - cdotu - compute the dot product of two vectors x and y.

cdotui - cdotci - Compute the complex conjugated indexed dot product.

cellmm - cellmm - Ellpack format matrix-matrix multiply

cellsm - cellsm - Ellpack format triangular solve

cfft2b - cfft2b - compute a periodic sequence from its Fourier coefficients. The xFFT operations are unnormalized, so a call of xFFT2F followed by a call of xFFT2B will multiply the input sequence by M*N.

cfft2f - cfft2f - compute the Fourier coefficients of a periodic sequence. The xFFT operations are unnormalized, so a call of xFFT2F followed by a call of xFFT2B will multiply the input sequence by M*N.

cfft2i - cfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

cfft3b - cfft3b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of CFFT3F followed by a call of CFFT3B will multiply the input sequence by M*N*K.

cfft3f - cfft3f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of CFFT3F followed by a call of CFFT3B will multiply the input sequence by M*N*K.

cfft3i - cfft3i - initialize the array WSAVE, which is used in both CFFT3F and CFFT3B.

cfftb - cfftb - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of CFFTF followed by a call of CFFTB will multiply the input sequence by N.

cfftc - cfftc - initialize the trigonometric weight and factor tables or compute the Fast Fourier transform (forward or inverse) of a complex sequence.

cfftc2 - cfftc2 - initialize the trigonometric weight and factor tables or compute the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional complex array.

cfftc3 - cfftc3 - initialize the trigonometric weight and factor tables or compute the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional complex array.

cfftcm - cfftcm - initialize the trigonometric weight and factor tables or compute the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional complex array.

cfftf - cfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of CFFTF followed by a call of CFFTB will multiply the input sequence by N.

cffti - cffti - initialize the array WSAVE, which is used in both CFFTF and CFFTB.

cfftopt - cfftopt - compute the length of the closest fast FFT

cffts - cffts - initialize the trigonometric weight and factor tables or compute the inverse Fast Fourier Transform of a complex sequence as follows.

cffts2 - cffts2 - initialize the trigonometric weight and factor tables or compute the two-dimensional inverse Fast Fourier Transform of a two-dimensional complex array.

cffts3 - cffts3 - initialize the trigonometric weight and factor tables or compute the three-dimensional inverse Fast Fourier Transform of a three-dimensional complex array.

cfftsm - cfftsm - initialize the trigonometric weight and factor tables or compute the one-dimensional inverse Fast Fourier Transform of a set of complex data sequences stored in a two-dimensional array.

cgbbrd - cgbbrd - reduce a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation

cgbcon - cgbcon - estimate the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm,

cgbequ - cgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

cgbmv - cgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

cgbrfs - cgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

cgbsv - cgbsv - compute the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

cgbsvx - cgbsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

cgbtf2 - cgbtf2 - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

cgbtrf - cgbtrf - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

cgbtrs - cgbtrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF

cgebak - cgebak - form the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL

cgebal - cgebal - balance a general complex matrix A

cgebrd - cgebrd - reduce a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation

cgecon - cgecon - estimate the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF

cgeequ - cgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

cgees - cgees - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

cgeesx - cgeesx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

cgeev - cgeev - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

cgeevx - cgeevx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

cgegs - cgegs - routine is deprecated and has been replaced by routine CGGES

cgegv - cgegv - routine is deprecated and has been replaced by routine CGGEV

cgehrd - cgehrd - reduce a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation

cgelqf - cgelqf - compute an LQ factorization of a complex M-by-N matrix A

cgels - cgels - solve overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A

cgelsd - cgelsd - compute the minimum-norm solution to a real linear least squares problem

cgelss - cgelss - compute the minimum norm solution to a complex linear least squares problem

cgelsx - cgelsx - routine is deprecated and has been replaced by routine CGELSY

cgelsy - cgelsy - compute the minimum-norm solution to a complex linear least squares problem

cgemm - cgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

cgemv - cgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

cgeqlf - cgeqlf - compute a QL factorization of a complex M-by-N matrix A

cgeqp3 - cgeqp3 - compute a QR factorization with column pivoting of a matrix A

cgeqpf - cgeqpf - routine is deprecated and has been replaced by routine CGEQP3

cgeqrf - cgeqrf - compute a QR factorization of a complex M-by-N matrix A

cgerc - cgerc - perform the rank 1 operation A := alpha*x*conjg(y') + A

cgerfs - cgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

cgerqf - cgerqf - compute an RQ factorization of a complex M-by-N matrix A

cgeru - cgeru - perform the rank 1 operation A := alpha*x*y' + A

cgesdd - cgesdd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method

cgesv - cgesv - compute the solution to a complex system of linear equations A * X = B,

cgesvd - cgesvd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors

cgesvx - cgesvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B,

cgetf2 - cgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

cgetrf - cgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

cgetri - cgetri - compute the inverse of a matrix using the LU factorization computed by CGETRF

cgetrs - cgetrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF

cggbak - cggbak - form the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL

cggbal - cggbal - balance a pair of general complex matrices (A,B)

cgges - cgges - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR)

cggesx - cggesx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T),

cggev - cggev - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

cggevx - cggevx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

cggglm - cggglm - solve a general Gauss-Markov linear model (GLM) problem

cgghrd - cgghrd - reduce a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular

cgglse - cgglse - solve the linear equality-constrained least squares (LSE) problem

cggqrf - cggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

cggrqf - cggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

cggsvd - cggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B

cggsvp - cggsvp - compute unitary matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

cgssco - cgssco - General sparse solver condition number estimate.

cgssda - cgssda - Deallocate working storage for the general sparse solver.

cgssfa - cgssfa - General sparse solver numeric factorization.

cgssfs - cgssfs - General sparse solver one call interface.

cgssin - cgssin - Initialize the general sparse solver.

cgssor - cgssor - General sparse solver ordering and symbolic factorization.

cgssps - cgssps - Print general sparse solver statics.

cgssrp - cgssrp - Return permutation used by the general sparse solver.

cgsssl - cgsssl - Solve routine for the general sparse solver.

cgssuo - cgssuo - User supplied permutation for ordering used in the general sparse solver.

cgtcon - cgtcon - estimate the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF

cgthr - cgthr - Gathers specified elements from y into x.

cgthrz - cgthrz - Gather and zero.

cgtrfs - cgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

cgtsv - cgtsv - solve the equation A*X = B,

cgtsvx - cgtsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

cgttrf - cgttrf - compute an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges

cgttrs - cgttrs - solve one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B,

chbev - chbev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

chbevd - chbevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

chbevx - chbevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

chbgst - chbgst - reduce a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

chbgv - chbgv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

chbgvd - chbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

chbgvx - chbgvx - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

chbmv - chbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

chbtrd - chbtrd - reduce a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

checon - checon - estimate the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

cheev - cheev - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

cheevd - cheevd - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

cheevr - cheevr - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T

cheevx - cheevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

chegs2 - chegs2 - reduce a complex Hermitian-definite generalized eigenproblem to standard form

chegst - chegst - reduce a complex Hermitian-definite generalized eigenproblem to standard form

chegv - chegv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chegvd - chegvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chegvx - chegvx - compute selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chemm - chemm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

chemv - chemv - perform the matrix-vector operation y := alpha*A*x + beta*y

cher - cher - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

cher2 - cher2 - perform the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

cher2k - cher2k - perform one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C

cherfs - cherfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution

cherk - cherk - perform one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C

chesv - chesv - compute the solution to a complex system of linear equations A * X = B,

chesvx - chesvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

chetf2 - chetf2 - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

chetrd - chetrd - reduce a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

chetrf - chetrf - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

chetri - chetri - compute the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

chetrs - chetrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

chgeqz - chgeqz - implement a single-shift version of the QZ method for finding the generalized eigenvalues w(i)=ALPHA(i)/BETA(i) of the equation det(A-w(i) B) = 0 If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form (i.e., A and B are both upper triangular) by applying one unitary tranformation (usually called Q) on the left and another (usually called Z) on the right

chpcon - chpcon - estimate the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

chpev - chpev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage

chpevd - chpevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

chpevx - chpevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

chpgst - chpgst - reduce a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage

chpgv - chpgv - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chpgvd - chpgvd - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chpgvx - chpgvx - compute selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

chpmv - chpmv - perform the matrix-vector operation y := alpha*A*x + beta*y

chpr - chpr - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

chpr2 - chpr2 - perform the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

chprfs - chprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution

chpsv - chpsv - compute the solution to a complex system of linear equations A * X = B,

chpsvx - chpsvx - use the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices

chptrd - chptrd - reduce a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation

chptrf - chptrf - compute the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method

chptri - chptri - compute the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

chptrs - chptrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

chsein - chsein - use inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H

chseqr - chseqr - compute the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors

cjadmm - cjadmm - Jagged diagonal matrix-matrix multiply (modified Ellpack)

cjadrp - cjadrp - right permutation of a jagged diagonal matrix

cjadsm - cjadsm - Jagged-diagonal format triangular solve

clarz - clarz - applie a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right

clarzb - clarzb - applie a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right

clarzt - clarzt - form the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors

clatzm - clatzm - routine is deprecated and has been replaced by routine CUNMRZ

cosqb - cosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

cosqf - cosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

cosqi - cosqi - initialize the array WSAVE, which is used in both COSQF and COSQB.

cost - cost - compute the discrete Fourier cosine transform of an even sequence. The COST transforms are unnormalized inverses of themselves, so a call of COST followed by another call of COST will multiply the input sequence by 2 * (N-1).

costi - costi - initialize the array WSAVE, which is used in COST.

cpbcon - cpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

cpbequ - cpbequ - compute row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm)

cpbrfs - cpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution

cpbstf - cpbstf - compute a split Cholesky factorization of a complex Hermitian positive definite band matrix A

cpbsv - cpbsv - compute the solution to a complex system of linear equations A * X = B,

cpbsvx - cpbsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

cpbtf2 - cpbtf2 - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

cpbtrf - cpbtrf - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

cpbtrs - cpbtrs - solve a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

cpocon - cpocon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

cpoequ - cpoequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm)

cporfs - cporfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite,

cposv - cposv - compute the solution to a complex system of linear equations A * X = B,

cposvx - cposvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

cpotf2 - cpotf2 - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

cpotrf - cpotrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

cpotri - cpotri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

cpotrs - cpotrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

cppcon - cppcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

cppequ - cppequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

cpprfs - cpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is

Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution

cppsv - cppsv - compute the solution to a complex system of linear equations A * X = B,

cppsvx - cppsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

cpptrf - cpptrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format

cpptri - cpptri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

cpptrs - cpptrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

cptcon - cptcon - compute the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF

cpteqr - cpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor

cptrfs - cptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

cptsv - cptsv - compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

cptsvx - cptsvx - use the factorization A = L*D*L**H to compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

cpttrf - cpttrf - compute the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A

cpttrs - cpttrs - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

cptts2 - cptts2 - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

crot - crot - apply a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex

crotg - crotg - Construct a Given's plane rotation

cscal - cscal - Compute y := alpha * y

csctr - csctr - Scatters elements from x into y.

cskymm - cskymm - Skyline format matrix-matrix multiply

cskysm - cskysm - Skyline format triangular solve

cspcon - cspcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

csprfs - csprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

cspsv - cspsv - compute the solution to a complex system of linear equations A * X = B,

cspsvx - cspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

csptrf - csptrf - compute the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

csptri - csptri - compute the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

csptrs - csptrs - solve a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

csrot - csrot - Apply a plane rotation.

csscal - csscal - Compute y := alpha * y

cstedc - cstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

cstegr - cstegr - Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

cstein - cstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

csteqr - csteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

cstsv - cstsv - compute the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix

csttrf - csttrf - compute the factorization of a complex Hermitian tridiagonal matrix A

csttrs - csttrs - computes the solution to a complex system of linear equations A * X = B

cswap - cswap - Exchange vectors x and y.

csycon - csycon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

csymm - csymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

csyr2k - csyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

csyrfs - csyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

csyrk - csyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

csysv - csysv - compute the solution to a complex system of linear equations A * X = B,

csysvx - csysvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

csytf2 - csytf2 - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

csytrf - csytrf - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

csytri - csytri - compute the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

csytrs - csytrs - solve a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

ctbcon - ctbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

ctbmv - ctbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ctbrfs - ctbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

ctbsv - ctbsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ctbtrs - ctbtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ctgevc - ctgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B)

ctgexc - ctgexc - reorder the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST

ctgsen - ctgsen - reorder the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B)

ctgsja - ctgsja - compute the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B

ctgsna - ctgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B)

ctgsyl - ctgsyl - solve the generalized Sylvester equation

ctpcon - ctpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

ctpmv - ctpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ctprfs - ctprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

ctpsv - ctpsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ctptri - ctptri - compute the inverse of a complex upper or lower triangular matrix A stored in packed format

ctptrs - ctptrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ctrans - ctrans - transpose and scale source matrix

ctrcon - ctrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

ctrevc - ctrevc - compute some or all of the right and/or left eigenvectors of a complex upper triangular matrix T

ctrexc - ctrexc - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST

ctrmm - ctrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

ctrmv - ctrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ctrrfs - ctrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

ctrsen - ctrsen - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace

ctrsm - ctrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

ctrsna - ctrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary)

ctrsv - ctrsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ctrsyl - ctrsyl - solve the complex Sylvester matrix equation

ctrti2 - ctrti2 - compute the inverse of a complex upper or lower triangular matrix

ctrtri - ctrtri - compute the inverse of a complex upper or lower triangular matrix A

ctrtrs - ctrtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ctzrqf - ctzrqf - routine is deprecated and has been replaced by routine CTZRZF

ctzrzf - ctzrzf - reduce the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations

cung2l - cung2l - generate an m by n complex matrix Q with orthonormal columns,

cung2r - cung2r - generate an m by n complex matrix Q with orthonormal columns,

cungbr - cungbr - generate one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form

cunghr - cunghr - generate a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD

cungl2 - cungl2 - generate an m-by-n complex matrix Q with orthonormal rows,

cunglq - cunglq - generate an M-by-N complex matrix Q with orthonormal rows,

cungql - cungql - generate an M-by-N complex matrix Q with orthonormal columns,

cungqr - cungqr - generate an M-by-N complex matrix Q with orthonormal columns,

cungr2 - cungr2 - generate an m by n complex matrix Q with orthonormal rows,

cungrq - cungrq - generate an M-by-N complex matrix Q with orthonormal rows,

cungtr - cungtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD

cunmbr - cunmbr - VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmhr - cunmhr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunml2 - cunml2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

cunmlq - cunmlq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmql - cunmql - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmqr - cunmqr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmr2 - cunmr2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

cunmrq - cunmrq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmrz - cunmrz - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cunmtr - cunmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cupgtr - cupgtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors H(i) of order n, as returned by CHPTRD using packed storage

cupmtr - cupmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

cvbrmm - cvbrmm - variable block sparse row format matrix-matrix multiply

cvbrsm - cvbrsm - variable block sparse row format triangular solve

cvmul - cvmul - compute the scaled product of complex vectors

dasum - dasum - Return the sum of the absolute values of a vector x.

daxpy - daxpy - compute y := alpha * x + y

daxpyi - daxpyi - Compute y := alpha * x + y

dbcomm - dbcomm - block coordinate matrix-matrix multiply

dbdimm - dbdimm - block diagonal format matrix-matrix multiply

dbdism - dbdism - block diagonal format triangular solve

dbdsdc - dbdsdc - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B

dbdsqr - dbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

dbelmm - dbelmm - block Ellpack format matrix-matrix multiply

dbelsm - dbelsm - block Ellpack format triangular solve

dbscmm - dbscmm - block sparse column matrix-matrix multiply

dbscsm - dbscsm - block sparse column format triangular solve

dbsrmm - dbsrmm - block sparse row format matrix-matrix multiply

dbsrsm - dbsrsm - block sparse row format triangular solve

dcnvcor - dcnvcor - compute the convolution or correlation of real vectors

dcnvcor2 - dcnvcor2 - compute the convolution or correlation of real matrices

dcoomm - dcoomm - coordinate matrix-matrix multiply

dcopy - dcopy - Copy x to y

dcosqb - dcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

dcosqf - dcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

dcosqi - dcosqi - initialize the array WSAVE, which is used in both COSQF and COSQB.

dcost - dcost - compute the discrete Fourier cosine transform of an even sequence. The COST transforms are unnormalized inverses of themselves, so a call of COST followed by another call of COST will multiply the input sequence by 2 * (N-1).

dcosti - dcosti - initialize the array WSAVE, which is used in COST.

dcscmm - dcscmm - compressed sparse column format matrix-matrix multiply

dcscsm - dcscsm - compressed sparse column format triangular solve

dcsrmm - dcsrmm - compressed sparse row format matrix-matrix multiply

dcsrsm - dcsrsm - compressed sparse row format triangular solve

ddiamm - ddiamm - diagonal format matrix-matrix multiply

ddiasm - ddiasm - diagonal format triangular solve

ddisna - ddisna - compute the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix

ddot - ddot - compute the dot product of two vectors x and y.

ddoti - ddoti - Compute the indexed dot product.

dellmm - dellmm - Ellpack format matrix-matrix multiply

dellsm - dellsm - Ellpack format triangular solve

dezftb - dezftb - computes a periodic sequence from its Fourier coefficients. DEZFTB is a simplified but slower version of DFFTB.

dezftf - dezftf - computes the Fourier coefficients of a periodic sequence. DEZFTF is a simplified but slower version

of DFFTF.

dezfti - dezfti - initializes the array WSAVE, which is used in both DEZFTF and DEZFTB.

dfft2b - dfft2b - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFT2F followed by a call of DFFT2B will multiply the input sequence by M*N.

dfft2f - dfft2f - compute the Fourier coefficients of a periodic sequence. The DFFT operations are unnormalized, so a call of DFFT2F followed by a call of DFFT2B will multiply the input sequence by M*N.

dfft2i - dfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

dfft3b - dfft3b - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFT3F followed by a call of DFFT3B will multiply the input sequence by M*N*K.

dfft3f - dfft3f - compute the Fourier coefficients of a real periodic sequence. The DFFT operations are unnormalized, so a call of DFFT3F followed by a call of DFFT3B will multiply the input sequence by M*N*K.

dfft3i - dfft3i - initialize the array WSAVE, which is used in both DFFT3F and DFFT3B.

dfftb - dfftb - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFTF followed by a call of DFFTB will multiply the input sequence by N.

dfftf - dfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of DFFTF followed by a call of DFFTB will multiply the input sequence by N.

dffti - dffti - initialize the array WSAVE, which is used in both DFFTF and DFFTB.

dfftopt - dfftopt - compute the length of the closest fast FFT

dfftz - dfftz - initialize the trigonometric weight and factor tables or compute the forward Fast Fourier Transform of a double precision sequence.

dfftz2 - dfftz2 - initialize the trigonometric weight and factor tables or compute the two-dimensional forward Fast Fourier Transform of a two-dimensional double precision array.

dfftz3 - dfftz3 - initialize the trigonometric weight and factor tables or compute the three-dimensional forward Fast Fourier Transform of a three-dimensional double complex array.

dfftzm - dfftzm - initialize the trigonometric weight and factor tables or compute the one-dimensional forward Fast Fourier Transform of a set of double precision data sequences stored in a two-dimensional array.

dgbbrd - dgbbrd - reduce a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation

dgbcon - dgbcon - estimate the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm,

dgbequ - dgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

dgbmv - dgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

dgbrfs - dgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

dgbsv - dgbsv - compute the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

dgbsvx - dgbsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

dgbtf2 - dgbtf2 - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

dgbtrf - dgbtrf - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

dgbtrs - dgbtrs - solve a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF

dgebak - dgebak - form the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL

dgebal - dgebal - balance a general real matrix A

dgebrd - dgebrd - reduce a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation

dgecon - dgecon - estimate the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF

dgeequ - dgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

dgees - dgees - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

dgeesx - dgeesx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

dgeev - dgeev - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

dgeevx - dgeevx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

dgegs - dgegs - routine is deprecated and has been replaced by routine SGGES

dgegv - dgegv - routine is deprecated and has been replaced by routine SGGEV

dgehrd - dgehrd - reduce a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation

dgelqf - dgelqf - compute an LQ factorization of a real M-by-N matrix A

dgels - dgels - solve overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A

dgelsd - dgelsd - compute the minimum-norm solution to a real linear least squares problem

dgelss - dgelss - compute the minimum norm solution to a real linear least squares problem

dgelsx - dgelsx - routine is deprecated and has been replaced by routine SGELSY

dgelsy - dgelsy - compute the minimum-norm solution to a real linear least squares problem

dgemm - dgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

dgemv - dgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

dgeqlf - dgeqlf - compute a QL factorization of a real M-by-N matrix A

dgeqp3 - dgeqp3 - compute a QR factorization with column pivoting of a matrix A

dgeqpf - dgeqpf - routine is deprecated and has been replaced by routine SGEQP3

dgeqrf - dgeqrf - compute a QR factorization of a real M-by-N matrix A

dger - dger - perform the rank 1 operation A := alpha*x*y' + A

dgerfs - dgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

dgerqf - dgerqf - compute an RQ factorization of a real M-by-N matrix A

dgesdd - dgesdd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and right singular vectors

dgesv - dgesv - compute the solution to a real system of linear equations A * X = B,

dgesvd - dgesvd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors

dgesvx - dgesvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B,

dgetf2 - dgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

dgetrf - dgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

dgetri - dgetri - compute the inverse of a matrix using the LU factorization computed by SGETRF

dgetrs - dgetrs - solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF

dggbak - dggbak - form the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL

dggbal - dggbal - balance a pair of general real matrices (A,B)

dgges - dgges - compute for a pair of N-by-N real nonsymmetric matrices (A,B),

dggesx - dggesx - compute for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and,

dggev - dggev - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

dggevx - dggevx - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

dggglm - dggglm - solve a general Gauss-Markov linear model (GLM) problem

dgghrd - dgghrd - reduce a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular

dgglse - dgglse - solve the linear equality-constrained least squares (LSE) problem

dggqrf - dggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

dggrqf - dggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

dggsvd - dggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B

dggsvp - dggsvp - compute orthogonal matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

dgssco - dgssco - General sparse solver condition number estimate.

dgssda - dgssda - Deallocate working storage for the general sparse solver.

dgssfa - dgssfa - General sparse solver numeric factorization.

dgssfs - dgssfs - General sparse solver one call interface.

dgssin - dgssin - Initialize the general sparse solver.

dgssor - dgssor - General sparse solver ordering and symbolic factorization.

dgssps - dgssps - Print general sparse solver statics.

dgssrp - dgssrp - Return permutation used by the general sparse solver.

dgsssl - dgsssl - Solve routine for the general sparse solver.

dgssuo - dgssuo - User supplied permutation for ordering used in the general sparse solver.

dgtcon - dgtcon - estimate the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF

dgthr - dgthr - Gathers specified elements from y into x.

dgthrz - dgthrz - Gather and zero.

dgtrfs - dgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

dgtsv - dgtsv - solve the equation A*X = B,

dgtsvx - dgtsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B,

dgttrf - dgttrf - compute an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges

dgttrs - dgttrs - solve one of the systems of equations A*X = B or A'*X = B,

dhgeqz - dhgeqz - implement a single-/double-shift version of the QZ method for finding the generalized eigenvalues w(j)=(ALPHAR(j) + i*ALPHAI(j))/BETAR(j) of the equation det(A-w(i) B) = 0 In addition, the pair A,B may be reduced to generalized Schur form

dhsein - dhsein - use inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H

dhseqr - dhseqr - compute the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors

djadmm - djadmm - Jagged diagonal matrix-matrix multiply (modified Ellpack)

djadrp - djadrp - right permutation of a jagged diagonal matrix

djadsm - djadsm - Jagged-diagonal format triangular solve

dlagtf - dlagtf - factorize the matrix (T-lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as T-lambda*I = PLU

dlamrg - dlamrg - will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order

dlarz - dlarz - applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right

dlarzb - dlarzb - applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right

dlarzt - dlarzt - form the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors

dlasrt - dlasrt - the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D')

dlatzm - dlatzm - routine is deprecated and has been replaced by routine SORMRZ

dnrm2 - dnrm2 - Return the Euclidian norm of a vector.

dopgtr - dopgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors H(i) of order n, as returned by SSPTRD using packed storage

dopmtr - dopmtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dorg2l - dorg2l - generate an m by n real matrix Q with orthonormal columns,

dorg2r - dorg2r - generate an m by n real matrix Q with orthonormal columns,

dorgbr - dorgbr - generate one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form

dorghr - dorghr - generate a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD

dorgl2 - dorgl2 - generate an m by n real matrix Q with orthonormal rows,

dorglq - dorglq - generate an M-by-N real matrix Q with orthonormal rows,

dorgql - dorgql - generate an M-by-N real matrix Q with orthonormal columns,

dorgqr - dorgqr - generate an M-by-N real matrix Q with orthonormal columns,

dorgr2 - dorgr2 - generate an m by n real matrix Q with orthonormal rows,

dorgrq - dorgrq - generate an M-by-N real matrix Q with orthonormal rows,

dorgtr - dorgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD

dormbr - dormbr - VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormhr - dormhr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormlq - dormlq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormql - dormql - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormqr - dormqr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormrq - dormrq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormrz - dormrz - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dormtr - dormtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

dpbcon - dpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

dpbequ - dpbequ - compute row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm)

dpbrfs - dpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution

dpbstf - dpbstf - compute a split Cholesky factorization of a real symmetric positive definite band matrix A

dpbsv - dpbsv - compute the solution to a real system of linear equations A * X = B,

dpbsvx - dpbsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

dpbtf2 - dpbtf2 - compute the Cholesky factorization of a real symmetric positive definite band matrix A

dpbtrf - dpbtrf - compute the Cholesky factorization of a real symmetric positive definite band matrix A

dpbtrs - dpbtrs - solve a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

dpocon - dpocon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

dpoequ - dpoequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A and reduce its condition number (with respect to the two-norm)

dporfs - dporfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite,

dposv - dposv - compute the solution to a real system of linear equations A * X = B,

dposvx - dposvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

dpotf2 - dpotf2 - compute the Cholesky factorization of a real symmetric positive definite matrix A

dpotrf - dpotrf - compute the Cholesky factorization of a real symmetric positive definite matrix A

dpotri - dpotri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

dpotrs - dpotrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

dppcon - dppcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

dppequ - dppequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

dpprfs - dpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution

dppsv - dppsv - compute the solution to a real system of linear equations A * X = B,

dppsvx - dppsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

dpptrf - dpptrf - compute the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format

dpptri - dpptri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

dpptrs - dpptrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

dptcon - dptcon - compute the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF

dpteqr - dpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor

dptrfs - dptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

dptsv - dptsv - compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

dptsvx - dptsvx - use the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

dpttrf - dpttrf - compute the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A

dpttrs - dpttrs - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

dptts2 - dptts2 - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

dqdota - dqdota - compute a double precision constant plus an extended precision constant plus the extended precision dot product of two double precision vectors x and y.

dqdoti - dqdoti - compute a constant plus the extended precision dot product of two double precision vectors x and y.

drot - drot - Apply a Given's rotation constructed by SROTG.

drotg - drotg - Construct a Given's plane rotation

droti - droti - Apply an indexed Givens rotation.

drotm - drotm - Apply a Gentleman's modified Given's rotation constructed by SROTMG.

drotmg - drotmg - Construct a Gentleman's modified Given's plane rotation

dsbev - dsbev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

dsbevd - dsbevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

dsbevx - dsbevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

dsbgst - dsbgst - reduce a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

dsbgv - dsbgv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

dsbgvd - dsbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

dsbgvx - dsbgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

dsbmv - dsbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

dsbtrd - dsbtrd - reduce a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

dscal - dscal - Compute y := alpha * y

dsctr - dsctr - Scatters elements from x into y.

dsdot - dsdot - compute the double precision dot product of two single precision vectors x and y.

dsecnd - dsecnd - return the user time for a process in seconds

dsinqb - dsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

dsinqf - dsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

dsinqi - dsinqi - initialize the array xWSAVE, which is used in both SINQF and SINQB.

dsint - dsint - compute the discrete Fourier sine transform of an odd sequence. The SINT transforms are unnormalized inverses of themselves, so a call of SINT followed by another call of SINT will multiply the input sequence by 2 * (N+1).

dsinti - dsinti - initialize the array WSAVE, which is used in subroutine SINT.

dskymm - dskymm - Skyline format matrix-matrix multiply

dskysm - dskysm - Skyline format triangular solve

dspcon - dspcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

dspev - dspev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

dspevd - dspevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

dspevx - dspevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

dspgst - dspgst - reduce a real symmetric-definite generalized eigenproblem to standard form, using packed storage

dspgv - dspgv - compute all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dspgvd - dspgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dspgvx - dspgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dspmv - dspmv - perform the matrix-vector operation y := alpha*A*x + beta*y

dspr - dspr - perform the symmetric rank 1 operation A := alpha*x*x' + A

dspr2 - dspr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

dsprfs - dsprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

dspsv - dspsv - compute the solution to a real system of linear equations A * X = B,

dspsvx - dspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

dsptrd - dsptrd - reduce a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation

dsptrf - dsptrf - compute the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

dsptri - dsptri - compute the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

dsptrs - dsptrs - solve a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

dstebz - dstebz - compute the eigenvalues of a symmetric tridiagonal matrix T

dstedc - dstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

dstegr - dstegr - (a) Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

dstein - dstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

dsteqr - dsteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

dsterf - dsterf - compute all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm

dstev - dstev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

dstevd - dstevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix

dstevr - dstevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

dstevx - dstevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

dstsv - dstsv - compute the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix

dsttrf - dsttrf - compute the factorization of a symmetric tridiagonal matrix A

dsttrs - dsttrs - computes the solution to a real system of linear equations A * X = B

dswap - dswap - Exchange vectors x and y.

dsycon - dsycon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

dsyev - dsyev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

dsyevd - dsyevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

dsyevr - dsyevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

dsyevx - dsyevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

dsygs2 - dsygs2 - reduce a real symmetric-definite generalized eigenproblem to standard form

dsygst - dsygst - reduce a real symmetric-definite generalized eigenproblem to standard form

dsygv - dsygv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dsygvd - dsygvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dsygvx - dsygvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

dsymm - dsymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

dsymv - dsymv - perform the matrix-vector operation y := alpha*A*x + beta*y

dsyr - dsyr - perform the symmetric rank 1 operation A := alpha*x*x' + A

dsyr2 - dsyr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

dsyr2k - dsyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

dsyrfs - dsyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

dsyrk - dsyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

dsysv - dsysv - compute the solution to a real system of linear equations A * X = B,

dsysvx - dsysvx - use the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B,

dsytd2 - dsytd2 - reduce a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

dsytf2 - dsytf2 - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

dsytrd - dsytrd - reduce a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation

dsytrf - dsytrf - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

dsytri - dsytri - compute the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

dsytrs - dsytrs - solve a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

dtbcon - dtbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

dtbmv - dtbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

dtbrfs - dtbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

dtbsv - dtbsv - solve one of the systems of equations A*x = b, or A'*x = b

dtbtrs - dtbtrs - solve a triangular system of the form A * X = B or A**T * X = B,

dtgevc - dtgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B)

dtgexc - dtgexc - reorder the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation (A, B) = Q * (A, B) * Z',

dtgsen - dtgsen - reorder the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B

dtgsja - dtgsja - compute the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B

dtgsna - dtgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z

dtgsyl - dtgsyl - solve the generalized Sylvester equation

dtpcon - dtpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

dtpmv - dtpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

dtprfs - dtprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

dtpsv - dtpsv - solve one of the systems of equations A*x = b, or A'*x = b

dtptri - dtptri - compute the inverse of a real upper or lower triangular matrix A stored in packed format

dtptrs - dtptrs - solve a triangular system of the form A * X = B or A**T * X = B,

dtrans - dtrans - transpose and scale source matrix

dtrcon - dtrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

dtrevc - dtrevc - compute some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T

dtrexc - dtrexc - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST

dtrmm - dtrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A)

dtrmv - dtrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

dtrrfs - dtrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

dtrsen - dtrsen - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T,

dtrsm - dtrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

dtrsna - dtrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal)

dtrsv - dtrsv - solve one of the systems of equations A*x = b, or A'*x = b

dtrsyl - dtrsyl - solve the real Sylvester matrix equation

dtrti2 - dtrti2 - compute the inverse of a real upper or lower triangular matrix

dtrtri - dtrtri - compute the inverse of a real upper or lower triangular matrix A

dtrtrs - dtrtrs - solve a triangular system of the form A * X = B or A**T * X = B,

dtzrqf - dtzrqf - routine is deprecated and has been replaced by routine STZRZF

dtzrzf - dtzrzf - reduce the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations

dvbrmm - dvbrmm - variable block sparse row format matrix-matrix multiply

dvbrsm - dvbrsm - variable block sparse row format triangular solve

dwiener - dwiener - perform Wiener deconvolution of two signals

dzasum - dzasum - Return the sum of the absolute values of a vector x.

dznrm2 - dznrm2 - Return the Euclidian norm of a vector.

ezfftb - ezfftb - computes a periodic sequence from its Fourier coefficients. EZFFTB is a simplified but slower version of RFFTB.

ezfftf - ezfftf - computes the Fourier coefficients of a periodic sequence. EZFFTF is a simplified but slower version of RFFTF.

ezffti - ezffti - initializes the array WSAVE, which is used in both EZFFTF and EZFFTB.

icamax - icamax - return the index of the element with largest absolute value.

idamax - idamax - return the index of the element with largest absolute value.

ilaenv - The name of the calling subroutine, in either upper case or lower case.

isamax - isamax - return the index of the element with largest absolute value.

izamax - izamax - return the index of the element with largest absolute value.

lsame - lsame - returns .TRUE. if CA is the same letter as CB regardless of case

rfft2b - rfft2b - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFT2F followed by a call of RFFT2B will multiply the input sequence by M*N.

rfft2f - rfft2f - compute the Fourier coefficients of a periodic sequence. The RFFT operations are unnormalized, so a call of RFFT2F followed by a call of RFFT2B will multiply the input sequence by M*N.

rfft2i - rfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

rfft3b - rfft3b - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFT3F followed by a call of RFFT3B will multiply the input sequence by M*N*K.

rfft3f - rfft3f - compute the Fourier coefficients of a real periodic sequence. The RFFT operations are unnormalized, so a call of RFFT3F followed by a call of RFFT3B will multiply the input sequence by M*N*K.

rfft3i - rfft3i - initialize the array WSAVE, which is used in both RFFT3F and RFFT3B.

rfftb - rfftb - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFTF followed by a call of RFFTB will multiply the input sequence by N.

rfftf - rfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of RFFTF followed by a call of RFFTB will multiply the input sequence by N.

rffti - rffti - initialize the array WSAVE, which is used in both RFFTF and RFFTB.

rfftopt - rfftopt - compute the length of the closest fast FFT

sasum - sasum - Return the sum of the absolute values of a vector x.

saxpy - saxpy - compute y := alpha * x + y

saxpyi - saxpyi - Compute y := alpha * x + y

sbcomm - sbcomm - block coordinate matrix-matrix multiply

sbdimm - sbdimm - block diagonal format matrix-matrix multiply

sbdism - sbdism - block diagonal format triangular solve

sbdsdc - sbdsdc - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B

sbdsqr - sbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

sbelmm - sbelmm - block Ellpack format matrix-matrix multiply

sbelsm - sbelsm - block Ellpack format triangular solve

sbscmm - sbscmm - block sparse column matrix-matrix multiply

sbscsm - sbscsm - block sparse column format triangular solve

sbsrmm - sbsrmm - block sparse row format matrix-matrix multiply

sbsrsm - sbsrsm - block sparse row format triangular solve

scasum - scasum - Return the sum of the absolute values of a vector x.

scnrm2 - scnrm2 - Return the Euclidian norm of a vector.

scnvcor - scnvcor - compute the convolution or correlation of real vectors

scnvcor2 - scnvcor2 - compute the convolution or correlation of real matrices

scoomm - scoomm - coordinate matrix-matrix multiply

scopy - scopy - Copy x to y

scscmm - scscmm - compressed sparse column format matrix-matrix multiply

scscsm - scscsm - compressed sparse column format triangular solve

scsrmm - scsrmm - compressed sparse row format matrix-matrix multiply

scsrsm - scsrsm - compressed sparse row format triangular solve

sdiamm - sdiamm - diagonal format matrix-matrix multiply

sdiasm - sdiasm - diagonal format triangular solve

sdisna - sdisna - compute the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix

sdot - sdot - compute the dot product of two vectors x and y.

sdoti - sdoti - Compute the indexed dot product.

sdsdot - sdsdot - compute a constant plus the double precision dot product of two single precision vectors x and y

second - second - return the user time for a process in seconds

sellmm - sellmm - Ellpack format matrix-matrix multiply

sellsm - sellsm - Ellpack format triangular solve

sfftc - sfftc - initialize the trigonometric weight and factor tables or compute the forward Fast Fourier Transform of a real sequence.

sfftc2 - sfftc2 - initialize the trigonometric weight and factor tables or compute the two-dimensional forward Fast Fourier Transform of a two-dimensional real array.

sfftc3 - sfftc3 - initialize the trigonometric weight and factor tables or compute the three-dimensional forward Fast Fourier Transform of a three-dimensional complex array.

sfftcm - sfftcm - initialize the trigonometric weight and factor tables or compute the one-dimensional forward Fast Fourier Transform of a set of real data sequences stored in a two-dimensional array.

sgbbrd - sgbbrd - reduce a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation

sgbcon - sgbcon - estimate the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm,

sgbequ - sgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

sgbmv - sgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

sgbrfs - sgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

sgbsv - sgbsv - compute the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

sgbsvx - sgbsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

sgbtf2 - sgbtf2 - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

sgbtrf - sgbtrf - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

sgbtrs - sgbtrs - solve a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF

sgebak - sgebak - form the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL

sgebal - sgebal - balance a general real matrix A

sgebrd - sgebrd - reduce a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation

sgecon - sgecon - estimate the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF

sgeequ - sgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

sgees - sgees - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

sgeesx - sgeesx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

sgeev - sgeev - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

sgeevx - sgeevx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

sgegs - sgegs - routine is deprecated and has been replaced by routine SGGES

sgegv - sgegv - routine is deprecated and has been replaced by routine SGGEV

sgehrd - sgehrd - reduce a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation

sgelqf - sgelqf - compute an LQ factorization of a real M-by-N matrix A

sgels - sgels - solve overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A

sgelsd - sgelsd - compute the minimum-norm solution to a real linear least squares problem

sgelss - sgelss - compute the minimum norm solution to a real linear least squares problem

sgelsx - sgelsx - routine is deprecated and has been replaced by routine SGELSY

sgelsy - sgelsy - compute the minimum-norm solution to a real linear least squares problem

sgemm - sgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

sgemv - sgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

sgeqlf - sgeqlf - compute a QL factorization of a real M-by-N matrix A

sgeqp3 - sgeqp3 - compute a QR factorization with column pivoting of a matrix A

sgeqpf - sgeqpf - routine is deprecated and has been replaced by routine SGEQP3

sgeqrf - sgeqrf - compute a QR factorization of a real M-by-N matrix A

sger - sger - perform the rank 1 operation A := alpha*x*y' + A

sgerfs - sgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

sgerqf - sgerqf - compute an RQ factorization of a real M-by-N matrix A

sgesdd - sgesdd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally

computing the left and right singular vectors

sgesv - sgesv - compute the solution to a real system of linear equations A * X = B,

sgesvd - sgesvd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors

sgesvx - sgesvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B,

sgetf2 - sgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

sgetrf - sgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

sgetri - sgetri - compute the inverse of a matrix using the LU factorization computed by SGETRF

sgetrs - sgetrs - solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF

sggbak - sggbak - form the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL

sggbal - sggbal - balance a pair of general real matrices (A,B)

sgges - sgges - compute for a pair of N-by-N real nonsymmetric matrices (A,B),

sggesx - sggesx - compute for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and,

sggev - sggev - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

sggevx - sggevx - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

sggglm - sggglm - solve a general Gauss-Markov linear model (GLM) problem

sgghrd - sgghrd - reduce a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular

sgglse - sgglse - solve the linear equality-constrained least squares (LSE) problem

sggqrf - sggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

sggrqf - sggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

sggsvd - sggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B

sggsvp - sggsvp - compute orthogonal matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

sgssco - sgssco - General sparse solver condition number estimate.

sgssda - sgssda - Deallocate working storage for the general sparse solver.

sgssfa - sgssfa - General sparse solver numeric factorization.

sgssfs - sgssfs - General sparse solver one call interface.

sgssin - sgssin - Initialize the general sparse solver.

sgssor - sgssor - General sparse solver ordering and symbolic factorization.

sgssps - sgssps - Print general sparse solver statics.

sgssrp - sgssrp - Return permutation used by the general sparse solver.

sgsssl - sgsssl - Solve routine for the general sparse solver.

sgssuo - sgssuo - User supplied permutation for ordering used in the general sparse solver.

sgtcon - sgtcon - estimate the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF

sgthr - sgthr - Gathers specified elements from y into x.

sgthrz - sgthrz - Gather and zero.

sgtrfs - sgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

sgtsv - sgtsv - solve the equation A*X = B,

sgtsvx - sgtsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B,

sgttrf - sgttrf - compute an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges

sgttrs - sgttrs - solve one of the systems of equations A*X = B or A'*X = B,

shgeqz - shgeqz - implement a single-/double-shift version of the QZ method for finding the generalized eigenvalues w(j)=(ALPHAR(j) + i*ALPHAI(j))/BETAR(j) of the equation det(A-w(i) B) = 0 In addition, the pair A,B may be reduced to generalized Schur form

shsein - shsein - use inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H

shseqr - shseqr - compute the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors

sinqb - sinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

sinqf - sinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

sinqi - sinqi - initialize the array xWSAVE, which is used in both SINQF and SINQB.

sint - sint - compute the discrete Fourier sine transform of an odd sequence. The SINT transforms are unnormalized inverses of themselves, so a call of SINT followed by another call of SINT will multiply the input sequence by 2 * (N+1).

sinti - sinti - initialize the array WSAVE, which is used in subroutine SINT.

sjadmm - sjadmm - Jagged diagonal matrix-matrix multiply (modified Ellpack)

sjadrp - sjadrp - right permutation of a jagged diagonal matrix

sjadsm - sjadsm - Jagged-diagonal format triangular solve

slagtf - slagtf - factorize the matrix (T-lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as T-lambda*I = PLU

slamrg - slamrg - will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order

slarz - slarz - applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right

slarzb - slarzb - applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right

slarzt - slarzt - form the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors

slasrt - slasrt - the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D')

slatzm - slatzm - routine is deprecated and has been replaced by routine SORMRZ

snrm2 - snrm2 - Return the Euclidian norm of a vector.

sopgtr - sopgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors

H(i) of order n, as returned by SSPTRD using packed storage

sopmtr - sopmtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sorg2l - sorg2l - generate an m by n real matrix Q with orthonormal columns,

sorg2r - sorg2r - generate an m by n real matrix Q with orthonormal columns,

sorgbr - sorgbr - generate one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form

sorghr - sorghr - generate a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD

sorgl2 - sorgl2 - generate an m by n real matrix Q with orthonormal rows,

sorglq - sorglq - generate an M-by-N real matrix Q with orthonormal rows,

sorgql - sorgql - generate an M-by-N real matrix Q with orthonormal columns,

sorgqr - sorgqr - generate an M-by-N real matrix Q with orthonormal columns,

sorgr2 - sorgr2 - generate an m by n real matrix Q with orthonormal rows,

sorgrq - sorgrq - generate an M-by-N real matrix Q with orthonormal rows,

sorgtr - sorgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD

sormbr - sormbr - VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormhr - sormhr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormlq - sormlq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormql - sormql - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormqr - sormqr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormrq - sormrq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormrz - sormrz - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

sormtr - sormtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

spbcon - spbcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

spbequ - spbequ - compute row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm)

spbrfs - spbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution

spbstf - spbstf - compute a split Cholesky factorization of a real symmetric positive definite band matrix A

spbsv - spbsv - compute the solution to a real system of linear equations A * X = B,

spbsvx - spbsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

spbtf2 - spbtf2 - compute the Cholesky factorization of a real symmetric positive definite band matrix A

spbtrf - spbtrf - compute the Cholesky factorization of a real symmetric positive definite band matrix A

spbtrs - spbtrs - solve a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

spocon - spocon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

spoequ - spoequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A

and reduce its condition number (with respect to the two-norm)

sporfs - sporfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite,

sposv - sposv - compute the solution to a real system of linear equations A * X = B,

sposvx - sposvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

spotf2 - spotf2 - compute the Cholesky factorization of a real symmetric positive definite matrix A

spotrf - spotrf - compute the Cholesky factorization of a real symmetric positive definite matrix A

spotri - spotri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

spotrs - spotrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

sppcon - sppcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

sppequ - sppequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

spprfs - spprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution

sppsv - sppsv - compute the solution to a real system of linear equations A * X = B,

sppsvx - sppsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

spptrf - spptrf - compute the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format

spptri - spptri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

spptrs - spptrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

sptcon - sptcon - compute the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF

spteqr - spteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor

sptrfs - sptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

sptsv - sptsv - compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

sptsvx - sptsvx - use the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

spttrf - spttrf - compute the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A

spttrs - spttrs - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

sptts2 - sptts2 - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

srot - srot - Apply a Given's rotation constructed by SROTG.

srotg - srotg - Construct a Given's plane rotation

sroti - sroti - Apply an indexed Givens rotation.

srotm - srotm - Apply a Gentleman's modified Given's rotation constructed by SROTMG.

srotmg - srotmg - Construct a Gentleman's modified Given's plane rotation

ssbev - ssbev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

ssbevd - ssbevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

ssbevx - ssbevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

ssbgst - ssbgst - reduce a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

ssbgv - ssbgv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

ssbgvd - ssbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

ssbgvx - ssbgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

ssbmv - ssbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

ssbtrd - ssbtrd - reduce a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

sscal - sscal - Compute y := alpha * y

ssctr - ssctr - Scatters elements from x into y.

sskymm - sskymm - Skyline format matrix-matrix multiply

sskysm - sskysm - Skyline format triangular solve

sspcon - sspcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

sspev - sspev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

sspevd - sspevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

sspevx - sspevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

sspgst - sspgst - reduce a real symmetric-definite generalized eigenproblem to standard form, using packed storage

sspgv - sspgv - compute all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

sspgvd - sspgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

sspgvx - sspgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

sspmv - sspmv - perform the matrix-vector operation y := alpha*A*x + beta*y

sspr - sspr - perform the symmetric rank 1 operation A := alpha*x*x' + A

sspr2 - sspr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

ssprfs - ssprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

sspsv - sspsv - compute the solution to a real system of linear equations A * X = B,

sspsvx - sspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution

to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

ssptrd - ssptrd - reduce a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation

ssptrf - ssptrf - compute the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

ssptri - ssptri - compute the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

ssptrs - ssptrs - solve a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

sstebz - sstebz - compute the eigenvalues of a symmetric tridiagonal matrix T

sstedc - sstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

sstegr - sstegr - (a) Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

sstein - sstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

ssteqr - ssteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

ssterf - ssterf - compute all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm

sstev - sstev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

sstevd - sstevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix

sstevr - sstevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

sstevx - sstevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

sstsv - sstsv - compute the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix

ssttrf - ssttrf - compute the factorization of a symmetric tridiagonal matrix A

ssttrs - ssttrs - computes the solution to a real system of linear equations A * X = B

sswap - sswap - Exchange vectors x and y.

ssycon - ssycon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

ssyev - ssyev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

ssyevd - ssyevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

ssyevr - ssyevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

ssyevx - ssyevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

ssygs2 - ssygs2 - reduce a real symmetric-definite generalized eigenproblem to standard form

ssygst - ssygst - reduce a real symmetric-definite generalized eigenproblem to standard form

ssygv - ssygv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

ssygvd - ssygvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

ssygvx - ssygvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

ssymm - ssymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

ssymv - ssymv - perform the matrix-vector operation y := alpha*A*x + beta*y

ssyr - ssyr - perform the symmetric rank 1 operation A := alpha*x*x' + A

ssyr2 - ssyr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

ssyr2k - ssyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

ssyrfs - ssyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

ssyrk - ssyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

ssysv - ssysv - compute the solution to a real system of linear equations A * X = B,

ssysvx - ssysvx - use the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B,

ssytd2 - ssytd2 - reduce a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

ssytf2 - ssytf2 - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

ssytrd - ssytrd - reduce a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation

ssytrf - ssytrf - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

ssytri - ssytri - compute the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

ssytrs - ssytrs - solve a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

stbcon - stbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

stbmv - stbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

stbrfs - stbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

stbsv - stbsv - solve one of the systems of equations A*x = b, or A'*x = b

stbtrs - stbtrs - solve a triangular system of the form A * X = B or A**T * X = B,

stgevc - stgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B)

stgexc - stgexc - reorder the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation (A, B) = Q * (A, B) * Z',

stgsen - stgsen - reorder the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B

stgsja - stgsja - compute the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B

stgsna - stgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z

stgsyl - stgsyl - solve the generalized Sylvester equation

stpcon - stpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

stpmv - stpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

stprfs - stprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

stpsv - stpsv - solve one of the systems of equations A*x = b, or A'*x = b

stptri - stptri - compute the inverse of a real upper or lower triangular matrix A stored in packed format

stptrs - stptrs - solve a triangular system of the form A * X = B or A**T * X = B,

strans - strans - transpose and scale source matrix

strcon - strcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

strevc - strevc - compute some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T

strexc - strexc - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST

strmm - strmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A)

strmv - strmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

strrfs - strrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

strsen - strsen - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T,

strsm - strsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

strsna - strsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal)

strsv - strsv - solve one of the systems of equations A*x = b, or A'*x = b

strsyl - strsyl - solve the real Sylvester matrix equation

strti2 - strti2 - compute the inverse of a real upper or lower triangular matrix

strtri - strtri - compute the inverse of a real upper or lower triangular matrix A

strtrs - strtrs - solve a triangular system of the form A * X = B or A**T * X = B,

stzrqf - stzrqf - routine is deprecated and has been replaced by routine STZRZF

stzrzf - stzrzf - reduce the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations

sunperf_version - sunperf_version - gets library information .HP 1i SUBROUTINE SUNPERF_VERSION(VERSION, PATCH, UPDATE) .HP 1i INTEGER VERSION, PATCH, UPDATE .HP 1i

svbrmm - svbrmm - variable block sparse row format matrix-matrix multiply

svbrsm - svbrsm - variable block sparse row format triangular solve

swiener - swiener - perform Wiener deconvolution of two signals

use_threads - use_threads - set the upper bound on the number of threads that the calling thread wants used

using_threads - using_threads - returns the current Use number set by the USE_THREADS subroutine

vcfftb - vcfftb - compute a periodic sequence from its Fourier coefficients. The VCFFT operations are normalized, so a call of VCFFTF followed by a call of VCFFTB will return the original sequence.

vcfftf - vcfftf - compute the Fourier coefficients of a periodic sequence. The VCFFT operations are normalized, so a call of VCFFTF followed by a call of VCFFTB will return the original sequence.

vcffti - vcffti - initialize the array WSAVE, which is used in both VCFFTF and VCFFTB.

vcosqb - vcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

vcosqf - vcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

vcosqi - vcosqi - initialize the array WSAVE, which is used in both VCOSQF and VCOSQB.

vcost - vcost - compute the discrete Fourier cosine transform of an even sequence. The VCOST transform is normalized, so a call of VCOST followed by a call of VCOST will return the original sequence.

vcosti - vcosti - initialize the array WSAVE, which is used in VCOST.

vdcosqb - vdcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

vdcosqf - vdcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

vdcosqi - vdcosqi - initialize the array WSAVE, which is used in both VCOSQF and VCOSQB.

vdcost - vdcost - compute the discrete Fourier cosine transform of an even sequence. The VCOST transform is normalized, so a call of VCOST followed by a call of VCOST will return the original sequence.

vdcosti - vdcosti - initialize the array WSAVE, which is used in VCOST.

vdfftb - vdfftb - compute a periodic sequence from its Fourier coefficients. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

vdfftf - vdfftf - compute the Fourier coefficients of a periodic sequence. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

vdffti - vdffti - initialize the array WSAVE, which is used in both VRFFTF and VRFFTB.

vdsinqb - vdsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

vdsinqf - vdsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

vdsinqi - vdsinqi - initialize the array WSAVE, which is used in both VSINQF and VSINQB.

vdsint - vdsint - compute the discrete Fourier sine transform of an odd sequence. The VSINT transforms are unnormalized inverses of themselves, so a call of VSINT followed by another call of VSINT will multiply the input sequence by 2 * (N+1). The VSINT transforms are normalized, so a call of VSINT followed by a call of VSINT will return the original sequence.

vdsinti - vdsinti - initialize the array WSAVE, which is used in subroutine VSINT.

vrfftb - vrfftb - compute a periodic sequence from its Fourier coefficients. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

vrfftf - vrfftf - compute the Fourier coefficients of a periodic sequence. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

vrffti - vrffti - initialize the array WSAVE, which is used in both VRFFTF and VRFFTB.

vsinqb - vsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

vsinqf - vsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The

VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

vsinqi - vsinqi - initialize the array WSAVE, which is used in both VSINQF and VSINQB.

vsint - vsint - compute the discrete Fourier sine transform of an odd sequence. The VSINT transforms are unnormalized inverses of themselves, so a call of VSINT followed by another call of VSINT will multiply the input sequence by 2 * (N+1). The VSINT transforms are normalized, so a call of VSINT followed by a call of VSINT will return the original sequence.

vsinti - vsinti - initialize the array WSAVE, which is used in subroutine VSINT.

vzfftb - vzfftb - compute a periodic sequence from its Fourier coefficients. The VZFFT operations are normalized, so a call of VZFFTF followed by a call of VZFFTB will return the original sequence.

vzfftf - vzfftf - compute the Fourier coefficients of a periodic sequence. The VZFFT operations are normalized, so a call of VZFFTF followed by a call of VZFFTB will return the original sequence.

vzffti - vzffti - initialize the array WSAVE, which is used in both VZFFTF and VZFFTB.

zaxpy - zaxpy - compute y := alpha * x + y

zaxpyi - zaxpyi - Compute y := alpha * x + y

zbcomm - zbcomm - block coordinate matrix-matrix multiply

zbdimm - zbdimm - block diagonal format matrix-matrix multiply

zbdism - zbdism - block diagonal format triangular solve

zbdsqr - zbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

zbelmm - zbelmm - block Ellpack format matrix-matrix multiply

zbelsm - zbelsm - block Ellpack format triangular solve

zbscmm - zbscmm - block sparse column matrix-matrix multiply

zbscsm - zbscsm - block sparse column format triangular solve

zbsrmm - zbsrmm - block sparse row format matrix-matrix multiply

zbsrsm - zbsrsm - block sparse row format triangular solve

zcnvcor - zcnvcor - compute the convolution or correlation of complex vectors

zcnvcor2 - zcnvcor2 - compute the convolution or correlation of complex matrices

zcoomm - zcoomm - coordinate matrix-matrix multiply

zcopy - zcopy - Copy x to y

zcscmm - zcscmm - compressed sparse column format matrix-matrix multiply

zcscsm - zcscsm - compressed sparse column format triangular solve

zcsrmm - zcsrmm - compressed sparse row format matrix-matrix multiply

zcsrsm - zcsrsm - compressed sparse row format triangular solve

zdiamm - zdiamm - diagonal format matrix-matrix multiply

zdiasm - zdiasm - diagonal format triangular solve

zdotc - zdotc - compute the dot product of two vectors conjg(x) and y.

zdotci - zdotci - Compute the complex conjugated indexed dot product.

zdotu - zdotu - compute the dot product of two vectors x and y.

zdotui - zdotui - Compute the complex unconjugated indexed dot product.

zdrot - zdrot - Apply a plane rotation.

zdscal - zdscal - Compute y := alpha * y

zellmm - zellmm - Ellpack format matrix-matrix multiply

zellsm - zellsm - Ellpack format triangular solve

zfft2b - zfft2b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFT2F followed by a call of ZFFT2B will multiply the input sequence by M*N.

zfft2f - zfft2f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFT2F followed by a call of ZFFT2B will multiply the input sequence by M*N.

zfft2i - zfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

zfft3b - zfft3b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFT3F followed by a call of ZFFT3B will multiply the input sequence by M*N*K.

zfft3f - zfft3f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFT3F followed by a call of ZFFT3B will multiply the input sequence by M*N*K.

zfft3i - zfft3i - initialize the array WSAVE, which is used in both ZFFT3F and ZFFT3B.

zfftb - zfftb - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFTF followed by a call of ZFFTB will multiply the input sequence by N.

zfftd - zfftd - initialize the trigonometric weight and factor tables or compute the inverse Fast Fourier Transform of a double complex sequence.

zfftd2 - zfftd2 - initialize the trigonometric weight and factor tables or compute the two-dimensional inverse Fast Fourier Transform of a two-dimensional double complex array.

zfftd3 - zfftd3 - initialize the trigonometric weight and factor tables or compute the three-dimensional inverse Fast Fourier Transform of a three-dimensional double complex array.

zfftdm - zfftdm - initialize the trigonometric weight and factor tables or compute the one-dimensional inverse Fast Fourier Transform of a set of double complex data sequences stored in a two-dimensional array.

zfftf - zfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFTF followed by a call of ZFFTB will multiply the input sequence by N.

zffti - zffti - initialize the array WSAVE, which is used in both ZFFTF and ZFFTB.

zfftopt - zfftopt - compute the length of the closest fast FFT

zfftz - zfftz - initialize the trigonometric weight and factor tables or compute the Fast Fourier transform (forward or inverse) of a double complex sequence.

zfftz2 - zfftz2 - initialize the trigonometric weight and factor tables or compute the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional double complex array.

zfftz3 - zfftz3 - initialize the trigonometric weight and factor tables or compute the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional double complex array.

zfftzm - zfftzm - initialize the trigonometric weight and factor tables or compute the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional double complex array.

zgbbrd - zgbbrd - reduce a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation

zgbcon - zgbcon - estimate the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm,

zgbequ - zgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

zgbmv - zgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

zgbrfs - zgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

zgbsv - zgbsv - compute the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

zgbsvx - zgbsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

zgbtf2 - zgbtf2 - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

zgbtrf - zgbtrf - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

zgbtrs - zgbtrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF

zgebak - zgebak - form the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL

zgebal - zgebal - balance a general complex matrix A

zgebrd - zgebrd - reduce a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation

zgecon - zgecon - estimate the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF

zgeequ - zgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

zgees - zgees - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

zgeesx - zgeesx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

zgeev - zgeev - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

zgeevx - zgeevx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

zgegs - zgegs - routine is deprecated and has been replaced by routine CGGES

zgegv - zgegv - routine is deprecated and has been replaced by routine CGGEV

zgehrd - zgehrd - reduce a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation

zgelqf - zgelqf - compute an LQ factorization of a complex M-by-N matrix A

zgels - zgels - solve overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A

zgelsd - zgelsd - compute the minimum-norm solution to a real linear least squares problem

zgelss - zgelss - compute the minimum norm solution to a complex linear least squares problem

zgelsx - zgelsx - routine is deprecated and has been replaced by routine CGELSY

zgelsy - zgelsy - compute the minimum-norm solution to a complex linear least squares problem

zgemm - zgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

zgemv - zgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

zgeqlf - zgeqlf - compute a QL factorization of a complex M-by-N matrix A

zgeqp3 - zgeqp3 - compute a QR factorization with column pivoting of a matrix A

zgeqpf - zgeqpf - routine is deprecated and has been replaced by routine CGEQP3

zgeqrf - zgeqrf - compute a QR factorization of a complex M-by-N matrix A

zgerc - zgerc - perform the rank 1 operation A := alpha*x*conjg(y') + A

zgerfs - zgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

zgerqf - zgerqf - compute an RQ factorization of a complex M-by-N matrix A

zgeru - zgeru - perform the rank 1 operation A := alpha*x*y' + A

zgesdd - zgesdd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method

zgesv - zgesv - compute the solution to a complex system of linear equations A * X = B,

zgesvd - zgesvd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors

zgesvx - zgesvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B,

zgetf2 - zgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

zgetrf - zgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

zgetri - zgetri - compute the inverse of a matrix using the LU factorization computed by CGETRF

zgetrs - zgetrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF

zggbak - zggbak - form the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL

zggbal - zggbal - balance a pair of general complex matrices (A,B)

zgges - zgges - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR)

zggesx - zggesx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T),

zggev - zggev - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

zggevx - zggevx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

zggglm - zggglm - solve a general Gauss-Markov linear model (GLM) problem

zgghrd - zgghrd - reduce a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular

zgglse - zgglse - solve the linear equality-constrained least squares (LSE) problem

zggqrf - zggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

zggrqf - zggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

zggsvd - zggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B

zggsvp - zggsvp - compute unitary matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

zgssco - zgssco - General sparse solver condition number estimate.

zgssda - zgssda - Deallocate working storage for the general sparse solver.

zgssfa - zgssfa - General sparse solver numeric factorization.

zgssfs - zgssfs - General sparse solver one call interface.

zgssin - zgssin - Initialize the general sparse solver.

zgssor - zgssor - General sparse solver ordering and symbolic factorization.

zgssps - zgssps - Print general sparse solver statics.

zgssrp - zgssrp - Return permutation used by the general sparse solver.

zgsssl - zgsssl - Solve routine for the general sparse solver.

zgssuo - zgssuo - User supplied permutation for ordering used in the general sparse solver.

zgtcon - zgtcon - estimate the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF

zgthr - zgthr - Gathers specified elements from y into x.

zgthrz - zgthrz - Gather and zero.

zgtrfs - zgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

zgtsv - zgtsv - solve the equation A*X = B,

zgtsvx - zgtsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

zgttrf - zgttrf - compute an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges

zgttrs - zgttrs - solve one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B,

zhbev - zhbev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

zhbevd - zhbevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

zhbevx - zhbevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

zhbgst - zhbgst - reduce a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

zhbgv - zhbgv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

zhbgvd - zhbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

zhbgvx - zhbgvx - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

zhbmv - zhbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

zhbtrd - zhbtrd - reduce a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

zhecon - zhecon - estimate the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

zheev - zheev - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

zheevd - zheevd - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

zheevr - zheevr - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T

zheevx - zheevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

zhegs2 - zhegs2 - reduce a complex Hermitian-definite generalized eigenproblem to standard form

zhegst - zhegst - reduce a complex Hermitian-definite generalized eigenproblem to standard form

zhegv - zhegv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhegvd - zhegvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhegvx - zhegvx - compute selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhemm - zhemm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

zhemv - zhemv - perform the matrix-vector operation y := alpha*A*x + beta*y

zher - zher - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

zher2 - zher2 - perform the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

zher2k - zher2k - perform one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C

zherfs - zherfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution

zherk - zherk - perform one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C

zhesv - zhesv - compute the solution to a complex system of linear equations A * X = B,

zhesvx - zhesvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

zhetf2 - zhetf2 - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

zhetrd - zhetrd - reduce a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

zhetrf - zhetrf - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

zhetri - zhetri - compute the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

zhetrs - zhetrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

zhgeqz - zhgeqz - implement a single-shift version of the QZ method for finding the generalized eigenvalues w(i)=ALPHA(i)/BETA(i) of the equation det(A-w(i) B) = 0 If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form (i.e., A and B are both upper triangular) by applying one unitary tranformation (usually called Q) on the left and another (usually called Z) on the right

zhpcon - zhpcon - estimate the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

zhpev - zhpev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage

zhpevd - zhpevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

zhpevx - zhpevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

zhpgst - zhpgst - reduce a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage

zhpgv - zhpgv - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized

Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhpgvd - zhpgvd - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhpgvx - zhpgvx - compute selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

zhpmv - zhpmv - perform the matrix-vector operation y := alpha*A*x + beta*y

zhpr - zhpr - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

zhpr2 - zhpr2 - perform the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

zhprfs - zhprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution

zhpsv - zhpsv - compute the solution to a complex system of linear equations A * X = B,

zhpsvx - zhpsvx - use the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices

zhptrd - zhptrd - reduce a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation

zhptrf - zhptrf - compute the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method

zhptri - zhptri - compute the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

zhptrs - zhptrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

zhsein - zhsein - use inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H

zhseqr - zhseqr - compute the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors

zjadmm - zjadmm - Jagged diagonal matrix-matrix multiply (modified Ellpack)

zjadrp - zjadrp - right permutation of a jagged diagonal matrix

zjadsm - zjadsm - Jagged-diagonal format triangular solve

zlarz - zlarz - applie a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right

zlarzb - zlarzb - applie a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right

zlarzt - zlarzt - form the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors

zlatzm - zlatzm - routine is deprecated and has been replaced by routine CUNMRZ

zpbcon - zpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

zpbequ - zpbequ - compute row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm)

zpbrfs - zpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution

zpbstf - zpbstf - compute a split Cholesky factorization of a complex Hermitian positive definite band matrix A

zpbsv - zpbsv - compute the solution to a complex system of linear equations A * X = B,

zpbsvx - zpbsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

zpbtf2 - zpbtf2 - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

zpbtrf - zpbtrf - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

zpbtrs - zpbtrs - solve a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

zpocon - zpocon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

zpoequ - zpoequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm)

zporfs - zporfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite,

zposv - zposv - compute the solution to a complex system of linear equations A * X = B,

zposvx - zposvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

zpotf2 - zpotf2 - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

zpotrf - zpotrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

zpotri - zpotri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

zpotrs - zpotrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

zppcon - zppcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

zppequ - zppequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

zpprfs - zpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution

zppsv - zppsv - compute the solution to a complex system of linear equations A * X = B,

zppsvx - zppsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

zpptrf - zpptrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format

zpptri - zpptri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

zpptrs - zpptrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

zptcon - zptcon - compute the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF

zpteqr - zpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor

zptrfs - zptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

zptsv - zptsv - compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

zptsvx - zptsvx - use the factorization A = L*D*L**H to compute the solution to a complex system of linear

equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

zpttrf - zpttrf - compute the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A

zpttrs - zpttrs - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

zptts2 - zptts2 - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

zrot - zrot - apply a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex

zrotg - zrotg - Construct a Given's plane rotation

zscal - zscal - Compute y := alpha * y

zsctr - zsctr - Scatters elements from x into y.

zskymm - zskymm - Skyline format matrix-matrix multiply

zskysm - zskysm - Skyline format triangular solve

zspcon - zspcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

zsprfs - zsprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

zspsv - zspsv - compute the solution to a complex system of linear equations A * X = B,

zspsvx - zspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

zsptrf - zsptrf - compute the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

zsptri - zsptri - compute the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

zsptrs - zsptrs - solve a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

zstedc - zstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

zstegr - zstegr - Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

zstein - zstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

zsteqr - zsteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

zstsv - zstsv - compute the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix

zsttrf - zsttrf - compute the factorization of a complex Hermitian tridiagonal matrix A

zsttrs - zsttrs - computes the solution to a complex system of linear equations A * X = B

zswap - zswap - Exchange vectors x and y.

zsycon - zsycon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

zsymm - zsymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

zsyr2k - zsyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C :=

alpha*A'*B + alpha*B'*A + beta*C

zsyrfs - zsyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

zsyrk - zsyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

zsysv - zsysv - compute the solution to a complex system of linear equations A * X = B,

zsysvx - zsysvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

zsytf2 - zsytf2 - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

zsytrf - zsytrf - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

zsytri - zsytri - compute the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

zsytrs - zsytrs - solve a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

ztbcon - ztbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

ztbmv - ztbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ztbrfs - ztbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

ztbsv - ztbsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ztbtrs - ztbtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ztgevc - ztgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B)

ztgexc - ztgexc - reorder the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST

ztgsen - ztgsen - reorder the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B)

ztgsja - ztgsja - compute the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B

ztgsna - ztgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B)

ztgsyl - ztgsyl - solve the generalized Sylvester equation

ztpcon - ztpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

ztpmv - ztpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ztprfs - ztprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

ztpsv - ztpsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ztptri - ztptri - compute the inverse of a complex upper or lower triangular matrix A stored in packed format

ztptrs - ztptrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ztrans - ztrans - transpose and scale source matrix

ztrcon - ztrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

ztrevc - ztrevc - compute some or all of the right and/or left eigenvectors of a complex upper triangular matrix T

ztrexc - ztrexc - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST

ztrmm - ztrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

ztrmv - ztrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

ztrrfs - ztrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

ztrsen - ztrsen - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace

ztrsm - ztrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

ztrsna - ztrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary)

ztrsv - ztrsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

ztrsyl - ztrsyl - solve the complex Sylvester matrix equation

ztrti2 - ztrti2 - compute the inverse of a complex upper or lower triangular matrix

ztrtri - ztrtri - compute the inverse of a complex upper or lower triangular matrix A

ztrtrs - ztrtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

ztzrqf - ztzrqf - routine is deprecated and has been replaced by routine CTZRZF

ztzrzf - ztzrzf - reduce the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations

zung2l - zung2l - generate an m by n complex matrix Q with orthonormal columns,

zung2r - zung2r - generate an m by n complex matrix Q with orthonormal columns,

zungbr - zungbr - generate one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form

zunghr - zunghr - generate a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD

zungl2 - zungl2 - generate an m-by-n complex matrix Q with orthonormal rows,

zunglq - zunglq - generate an M-by-N complex matrix Q with orthonormal rows,

zungql - zungql - generate an M-by-N complex matrix Q with orthonormal columns,

zungqr - zungqr - generate an M-by-N complex matrix Q with orthonormal columns,

zungr2 - zungr2 - generate an m by n complex matrix Q with orthonormal rows,

zungrq - zungrq - generate an M-by-N complex matrix Q with orthonormal rows,

zungtr - zungtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD

zunmbr - zunmbr - VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmhr - zunmhr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunml2 - zunml2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or

Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

zunmlq - zunmlq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmql - zunmql - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmqr - zunmqr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmr2 - zunmr2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

zunmrq - zunmrq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmrz - zunmrz - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zunmtr - zunmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zupgtr - zupgtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors H(i) of order n, as returned by CHPTRD using packed storage

zupmtr - zupmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

zvbrmm - zvbrmm - variable block sparse row format matrix-matrix multiply

zvbrsm - zvbrsm - variable block sparse row format triangular solve

zvmul - zvmul - compute the scaled product of complex vectors

---

NAME

available_threads - returns information about current thread usage

---

SYNOPSIS

```
SUBROUTINE AVAILABLE_THREADS( NTOTAL, NUSING)
INTEGER NTOTAL, NUSING

SUBROUTINE AVAILABLE_THREADS_64( NTOTAL, NUSING)
INTEGER*8 NTOTAL, NUSING
```

F95 INTERFACE

```
SUBROUTINE AVAILABLE_THREADS( NTOTAL, NUSING)
INTEGER :: NTOTAL, NUSING

SUBROUTINE AVAILABLE_THREADS_64( NTOTAL, NUSING)
INTEGER(8) :: NTOTAL, NUSING
```

C INTERFACE

#include <sunperf.h>

void available_threads(int *ntotal, int *nusing);

void available_threads_64(long *ntotal, long *nusing);

PURPOSE

available_threads threads returns NTOTAL, which is the total number of CPUs available to the job (generally the number of CPUs presently on-line in the partition), and NUSING, which is the sum of the current Use numbers for all threads specified in USE_THREADS. If NTOTAL < NUSING then the system is potentially overcommitted.

ARGUMENTS

- **NTOTAL (output)**
  Total number of CPUs available.
- **NUSING (output)**
  Sum of current Use numbers for all threads specified in USE_THREADS.

NAME

blas_dpermute - permutes a real (double precision) array in terms of the permutation vector P, output by dsortv

---

SYNOPSIS

```
SUBROUTINE BLAS_DPERMUTE( N, P, INCP, X, INCX)
INTEGER N, INCP, INCX
INTEGER P(*)
DOUBLE PRECISION X(*)

SUBROUTINE BLAS_DPERMUTE_64( N, P, INCP, X, INCX)
INTEGER*8 N, INCP, INCX
INTEGER*8 P(*)
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
SUBROUTINE PERMUTE( [N], P, [INCP], X, [INCX])
INTEGER :: N, INCP, INCX
INTEGER, DIMENSION(:) :: P
REAL(8), DIMENSION(:) :: X

SUBROUTINE PERMUTE_64( [N], P, [INCP], X, [INCX])
INTEGER(8) :: N, INCP, INCX
INTEGER(8), DIMENSION(:) :: P
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_dpermute(int n, int *p, int incp, double *x, int incx);

void blas_dpermute_64(long n, long *p, long incp, double *x, long incx);

ARGUMENTS

- **N (input)**
  If N < = 1, the subroutine returns without trying to permute X.
- **P (input)**
  vector defined follows the same conventions as that for DTYPE SORTV. It records the details of the interchanges of the elements of X during sorting. That is X = P*X. In current implementation, P contains the index of sorted X.
- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, the permutation is applied in the opposite direction. That is

  If INCP > 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
```

  If INCP < 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)).

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)

          = X(P((N-i)*|INCP|+1)).
```

- **X (input/output)**
  be permuted. Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, X will be permuted in a reverse way (see the description for INCP above).

SEE ALSO

blas_dsortv(3P), `blas_dsort(3P)`

NAME

blas_dsort - sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm

---

SYNOPSIS

```
SUBROUTINE BLAS_DSORT( SORT, N, X, INCX)
INTEGER SORT, N, INCX
DOUBLE PRECISION X(*)

SUBROUTINE BLAS_DSORT_64( SORT, N, X, INCX)
INTEGER*8 SORT, N, INCX
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
SUBROUTINE SORT( [SORT], [N], X, [INCX])
INTEGER :: SORT, N, INCX
REAL(8), DIMENSION(:) :: X

SUBROUTINE SORT_64( [SORT], [N], X, [INCX])
INTEGER(8) :: SORT, N, INCX
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_dsort(int sort, int n, double *x, int incx);

void blas_dsort_64(long sort, long n, double *x, long incx);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE
- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.
- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.

SEE ALSO

blas_dsortv(3P), `blas_dpermute(3P)`

NAME

blas_dsortv - sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector

---

SYNOPSIS

```
SUBROUTINE BLAS_DSORTV( SORT, N, X, INCX, P, INCP)
INTEGER SORT, N, INCX, INCP
INTEGER P(*)
DOUBLE PRECISION X(*)

SUBROUTINE BLAS_DSORTV_64( SORT, N, X, INCX, P, INCP)
INTEGER*8 SORT, N, INCX, INCP
INTEGER*8 P(*)
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
SUBROUTINE SORTV( [SORT], [N], X, [INCX], P, [INCP])
INTEGER :: SORT, N, INCX, INCP
INTEGER, DIMENSION(:) :: P
REAL(8), DIMENSION(:) :: X

SUBROUTINE SORTV_64( [SORT], [N], X, [INCX], P, [INCP])
INTEGER(8) :: SORT, N, INCX, INCP
INTEGER(8), DIMENSION(:) :: P
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_dsortv(int sort, int n, double *x, int incx, int *p, int incp);

void blas_dsortv_64(long sort, long n, double *x, long incx, long *p, long incp);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE
- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.
- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.
- **P (output)**
  vector recording the details of the interchanges of the elements of X during sorting. That is X = P*X. In this implementation, P contains the index of sorted X.
- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, store P(i) in reverse order. That is

  If INCP > 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
```

  If INCP < 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)

          = X(P((N-i)*|INCP|+1)).
```

SEE ALSO

blas_dsort(3P), `blas_dpermute(3P)`

NAME

blas_ipermute - permutes an integer array in terms of the permutation vector P, output by dsortv

---

SYNOPSIS

```
SUBROUTINE BLAS_IPERMUTE( N, P, INCP, X, INCX)
INTEGER N, INCP, INCX
INTEGER P(*), X(*)

SUBROUTINE BLAS_IPERMUTE_64( N, P, INCP, X, INCX)
INTEGER*8 N, INCP, INCX
INTEGER*8 P(*), X(*)
```

F95 INTERFACE

```
SUBROUTINE PERMUTE( [N], P, [INCP], X, [INCX])
INTEGER :: N, INCP, INCX
INTEGER, DIMENSION(:) :: P, X

SUBROUTINE PERMUTE_64( [N], P, [INCP], X, [INCX])
INTEGER(8) :: N, INCP, INCX
INTEGER(8), DIMENSION(:) :: P, X
```

C INTERFACE

#include <sunperf.h>

void blas_ipermute(int n, int *p, int incp, int *x, int incx);

void blas_ipermute_64(long n, long *p, long incp, long *x, long incx);

ARGUMENTS

- **N (input)**
  If N < = 1, the subroutine returns without trying to permute X.
- **P (input)**
  vector defined follows the same conventions as that for DTYPE SORTV. It records the details of the interchanges of the elements of X during sorting. That is X = P*X. In current implementation, P contains the index of sorted X.
- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, the permutation is applied in the opposite direction. That is

  If INCP > 0,

  ```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
  ```

  If INCP < 0,

  ```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)).

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)

          = X(P((N-i)*|INCP|+1)).
  ```
- **X (input/output)**
  to be permuted. Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, X will be permuted in a reverse way (see the description for INCP above).

SEE ALSO

blas_isortv(3P), `blas_isort(3P)`

NAME

blas_isort - sorts an integer vector X in increasing or decreasing order using quick sort algorithm

---

SYNOPSIS

```
SUBROUTINE BLAS_ISORT( SORT, N, X, INCX)
INTEGER SORT, N, INCX
INTEGER X(*)

SUBROUTINE BLAS_ISORT_64( SORT, N, X, INCX)
INTEGER*8 SORT, N, INCX
INTEGER*8 X(*)
```

F95 INTERFACE

```
SUBROUTINE SORT( [SORT], [N], X, [INCX])
INTEGER :: SORT, N, INCX
INTEGER, DIMENSION(:) :: X

SUBROUTINE SORT_64( [SORT], [N], X, [INCX])
INTEGER(8) :: SORT, N, INCX
INTEGER(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_isort(int sort, int n, int *x, int incx);

void blas_isort_64(long sort, long n, long *x, long incx);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE
- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.
- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.

SEE ALSO

blas_isortv(3P), `blas_ipermute(3P)`

NAME

blas_isortv - sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector

---

SYNOPSIS

```
SUBROUTINE BLAS_ISORTV( SORT, N, X, INCX, P, INCP)
INTEGER SORT, N, INCX, INCP
INTEGER X(*), P(*)

SUBROUTINE BLAS_ISORTV_64( SORT, N, X, INCX, P, INCP)
INTEGER*8 SORT, N, INCX, INCP
INTEGER*8 X(*), P(*)
```

F95 INTERFACE

```
SUBROUTINE SORTV( [SORT], [N], X, [INCX], P, [INCP])
INTEGER :: SORT, N, INCX, INCP
INTEGER, DIMENSION(:) :: X, P

SUBROUTINE SORTV_64( [SORT], [N], X, [INCX], P, [INCP])
INTEGER(8) :: SORT, N, INCX, INCP
INTEGER(8), DIMENSION(:) :: X, P
```

C INTERFACE

#include <sunperf.h>

void blas_isortv(int sort, int n, int *x, int incx, int *p, int incp);

void blas_isortv_64(long sort, long n, long *x, long incx, long *p, long incp);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE

- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.

- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required

- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.

- **P (output)**
  vector recording the details of the interchanges of the elements of X during sorting. That is X = P*X. In this implementation, P contains the index of sorted X.

- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, store P(i) in reverse order. That is

  If INCP > 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
```

  If INCP < 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)

          = X(P((N-i)*|INCP|+1)).
```

SEE ALSO

blas_isort(3P), `blas_ipermute(3P)`

NAME

blas_spermute - permutes a real array in terms of the permutation vector P, output by dsortv

---

SYNOPSIS

```
SUBROUTINE BLAS_SPERMUTE( N, P, INCP, X, INCX)
INTEGER N, INCP, INCX
INTEGER P(*)
REAL X(*)

SUBROUTINE BLAS_SPERMUTE_64( N, P, INCP, X, INCX)
INTEGER*8 N, INCP, INCX
INTEGER*8 P(*)
REAL X(*)
```

F95 INTERFACE

```
SUBROUTINE PERMUTE( [N], P, [INCP], X, [INCX])
INTEGER :: N, INCP, INCX
INTEGER, DIMENSION(:) :: P
REAL, DIMENSION(:) :: X

SUBROUTINE PERMUTE_64( [N], P, [INCP], X, [INCX])
INTEGER(8) :: N, INCP, INCX
INTEGER(8), DIMENSION(:) :: P
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_spermute(int n, int *p, int incp, float *x, int incx);

void blas_spermute_64(long n, long *p, long incp, float *x, long incx);

ARGUMENTS

- **N (input)**
  If N < = 1, the subroutine returns without trying to permute X.
- **P (input)**
  vector defined follows the same conventions as that for DTYPE SORTV. It records the details of the interchanges of the elements of X during sorting. That is X = P*X. In current implementation, P contains the index of sorted X.
- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, the permutation is applied in the opposite direction. That is

  If INCP > 0,

```
if INCX  > 0,

  sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

if INCX  < 0,

  sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
```

  If INCP < 0,

```
if INCX  > 0,

  sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)).

if INCX  < 0,

  sorted X((N-i)*|INCX|+1)

        = X(P((N-i)*|INCP|+1)).
```

- **X (input/output)**
  permuted. Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, X will be permuted in a reverse way (see the description for INCP above).

SEE ALSO

blas_ssortv(3P), `blas_ssort(3P)`

NAME

blas_ssort - sorts a real vector X in increasing or decreasing order using quick sort algorithm

---

SYNOPSIS

```
SUBROUTINE BLAS_SSORT( SORT, N, X, INCX)
INTEGER SORT, N, INCX
REAL X(*)

SUBROUTINE BLAS_SSORT_64( SORT, N, X, INCX)
INTEGER*8 SORT, N, INCX
REAL X(*)
```

F95 INTERFACE

```
SUBROUTINE SORT( [SORT], [N], X, [INCX])
INTEGER :: SORT, N, INCX
REAL, DIMENSION(:) :: X

SUBROUTINE SORT_64( [SORT], [N], X, [INCX])
INTEGER(8) :: SORT, N, INCX
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_ssort(int sort, int n, float *x, int incx);

void blas_ssort_64(long sort, long n, float *x, long incx);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE
- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.
- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.

SEE ALSO

blas_ssortv(3P), `blas_spermute(3P)`

NAME

blas_ssortv - sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector

---

SYNOPSIS

```
SUBROUTINE BLAS_SSORTV( SORT, N, X, INCX, P, INCP)
INTEGER SORT, N, INCX, INCP
INTEGER P(*)
REAL X(*)

SUBROUTINE BLAS_SSORTV_64( SORT, N, X, INCX, P, INCP)
INTEGER*8 SORT, N, INCX, INCP
INTEGER*8 P(*)
REAL X(*)
```

F95 INTERFACE

```
SUBROUTINE SORTV( [SORT], [N], X, [INCX], P, [INCP])
INTEGER :: SORT, N, INCX, INCP
INTEGER, DIMENSION(:) :: P
REAL, DIMENSION(:) :: X

SUBROUTINE SORTV_64( [SORT], [N], X, [INCX], P, [INCP])
INTEGER(8) :: SORT, N, INCX, INCP
INTEGER(8), DIMENSION(:) :: P
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

void blas_ssortv(int sort, int n, float *x, int incx, int *p, int incp);

void blas_ssortv_64(long sort, long n, float *x, long incx, long *p, long incp);

ARGUMENTS

- **SORT (input)**
  SORT = 0, descending

  SORT = 1, ascending

  SORT = other value, error

  SORT is default to 1 for F95 INTERFACE
- **N (input)**
  If N < = 1, the subroutine returns without trying to sort X.
- **X (input/output)**
  sorted

  Minimum size (N-1)*|INCX|+1 is required
- **INCX (input)**
  INCX must not be zero. INCX could be negative. If INCX < 0, change the sorting direction defined by SORT. That is

  If SORT = 0, let SORT = 1, INCX = |INCX|;

  If SORT = 1, let SORT = 0, INCX = |INCX|.
- **P (output)**
  vector recording the details of the interchanges of the elements of X during sorting. That is X = P*X. In this implementation, P contains the index of sorted X.
- **INCP (input)**
  INCP must not be zero. INCP could be negative. If INCP < 0, store P(i) in reverse order. That is

  If INCP > 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((i-1)*INCP+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)  = X(P((i-1)*INCP+1));
```

  If INCP < 0,

```
  if INCX  > 0,

    sorted X((i-1)*INCX+1)  = X(P((N-i)*|INCP|+1)),

  if INCX  < 0,

    sorted X((N-i)*|INCX|+1)

          = X(P((N-i)*|INCP|+1)).
```

SEE ALSO

blas_ssort(3P), `blas_spermute(3P)`

NAME

caxpy - compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE CAXPY( N, ALPHA, X, INCX, Y, INCY)
COMPLEX ALPHA
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE CAXPY_64( N, ALPHA, X, INCX, Y, INCY)
COMPLEX ALPHA
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE AXPY( [N], ALPHA, X, [INCX], Y, [INCY])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE AXPY_64( [N], ALPHA, X, [INCX], Y, [INCY])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void caxpy(int n, complex alpha, complex *x, int incx, complex *y, int incy);

void caxpy_64(long n, complex alpha, complex *x, long incx, complex *y, long incy);

PURPOSE

caxpy compute y := alpha * x + y where alpha is a scalar and x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  array of DIMENSION at least (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  array of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

caxpyi - Compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE CAXPYI(NZ, A, X, INDX, Y)

COMPLEX A
COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE CAXPYI_64(NZ, A, X, INDX, Y)

COMPLEX A
COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE AXPYI([NZ], [A], X, INDX, Y)

```
COMPLEX :: A
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE AXPYI_64([NZ], [A], X, INDX, Y)

COMPLEX :: A
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CAXPYI Compute y := alpha * x + y where alpha is a scalar, x is a sparse vector, and y is a vector in full storage form

```
do i = 1, n
  y(indx(i)) = alpha * x(i) + y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

A (input)

On entry, ALPHA specifies the scaling value. Unchanged on exit.

X (input)

Vector containing the values of the compressed form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

NAME

bcomm, sbcomm, dbcomm, cbcomm, zbcomm - block coordinate matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SBCOMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BJNDX, BNNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BNNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BJNDX(BNNZ)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBCOMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BJNDX, BNNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BNNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BJNDX(BNNZ)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBCOMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BJNDX, BNNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BNNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BJNDX(BNNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBCOMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BJNDX, BNNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BNNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BJNDX(BNNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

```
where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in block coordinate format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if the matrix is real.


MB            Number of block rows in matrix A

N             Number of columns in matrix C

KB            Number of block columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
```

```
                1 : no repeated indices

VAL()           scalar array of length LB*LB*BNNZ consisting of
                the non-zero block entries of A, in any order.
                Each block is stored in standard column-major form.

BINDX()         integer array of length BNNZ consisting of the
                block row indices of the block entries of A.

BJNDX()         integer array of length BNNZ consisting of the
                block column indices of the block entries of A.

BNNZ            number of block entries

LB              dimension of dense blocks composing A.

B()             rectangular array with first dimension LDB.

LDB             leading dimension of B

BETA            Scalar parameter

C()             rectangular array with first dimension LDC.

LDC             leading dimension of C

WORK()          scratch array of length LWORK. WORK is not
                referenced in the current version.


LWORK           length of WORK array. LWORK is not referenced
                in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

bdimm, sbdimm, dbdimm, cbdimm, zbdimm - block diagonal format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SBDIMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LB*LB*BLDA*NBDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBDIMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LB*LB*BLDA*NBDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBDIMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LB*LB*BLDA*NBDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBDIMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LB*LB*BLDA*NBDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)


=head1 DESCRIPTION
```

```
        C <- alpha op(A) B + beta C


where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in block diagonal format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                  ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix  is real.

MB            Number of block rows in matrix A

N             Number of columns in matrix C

KB            Number of block columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         two-dimensional LB*LB*BLDA-by-NBDIAG scalar array
              consisting of the NBDIAG nonzero block diagonal in
              any order. Each dense block is stored in standard
              column-major form.
```

BLDA	leading block dimension of VAL().
IBDIAG()	integer array of length NBDIAG consisting of the corresponding diagonal offsets of the non-zero block diagonals of A in VAL. Lower triangular block diagonals have negative offsets, the main block diagonal has offset 0, and upper triangular block diagonals have positive offset.
NBDIAG	the number of non-zero block diagonals in A.
LB	dimension of dense blocks composing A.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

bdism, sbdism, dbdism, cbdism, zbdism - block diagonal format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SBDISM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 REAL*4     ALPHA, BETA
 REAL*4     DV(MB*LB*LB), VAL(LB*LB*BLDA, NBDIAG), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE DBDISM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 REAL*8     ALPHA, BETA
 REAL*8     DV(MB*LB*LB), VAL(LB*LB*BLDA, NBDIAG), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE CBDISM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
 INTEGER*4  IBDIAG(NBDIAG)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(MB*LB*LB), VAL(LB*LB*BLDA, NBDIAG), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE ZBDISM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BLDA, IBDIAG, NBDIAG, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, NBDIAG, LB,
*           LDB, LDC, LWORK
```

```
 INTEGER*4  IBDIAG(NBDIAG)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(MB*LB*LB), VAL(LB*LB*BLDA, NBDIAG), B(LDB,*), C(LDC,*),
*           WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices,
D is a block diagonal matrix, A is a unit, or non-unit, upper or
lower triangular matrix represented in block diagonal format
and op(A) is one of

```
 op( A ) = inv(A) or  op( A ) = inv(A')  or  op( A ) =inv(conjg( A' ))
 (inv denotes matrix inverse,  ' indicates matrix transpose)
```

All blocks of A on the main diagonal MUST be triangular matrices.

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length MB*LB*LB containing the elements of the diagonal blocks of the matrix D. The size of each square block is LB-by-LB and each block is stored in standard column-major form.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general

1 : symmetric (A=A')
2 : Hermitian (A= CONJG(A'))
3 : Triangular
4 : Skew(Anti)-Symmetric (A=-A')
5 : Diagonal
6 : Skew-Hermitian (A= -CONJG(A'))
Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-identity blocks on the main diagonal
1 : identity diagonal block
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() Two-dimensional LB*LB*BLDA-by-NBDIAG scalar array consisting of the NBDIAG non-zero block diagonal. Each dense block is stored in standard column-major form.

BLDA Leading block dimension of VAL(). Should be greater than or equal to MB.

IBDIAG() integer array of length NBDIAG consisting of the corresponding diagonal offsets of the non-zero block diagonals of A in VAL. Lower triangular block diagonals have negative offsets, the main block diagonal has offset 0, and upper triangular block diagonals have positive offset. Elements of IBDIAG MUST be sorted in increasing order.

NBDIAG The number of non-zero block diagonals in A.

LB Dimension of dense blocks composing A.

B() Rectangular array with first dimension LDB.

LDB Leading dimension of B.

BETA Scalar parameter.

C() Rectangular array with first dimension LDC.

LDC Leading dimension of C.

WORK() scratch array of length LWORK.
On exit, if LWORK= -1, WORK(1) returns the optimum size of LWORK.

```
LWORK         length of WORK array. LWORK should be at least
              MB*LB.


              For good performance, LWORK should generally be larger.
              For optimum performance on multiple processors, LWORK
              >=MB*LB*N_CPUS where N_CPUS is the maximum number of
              processors available to the program.

              If LWORK=0, the routine is to allocate workspace needed.


              If LWORK = -1, then a workspace query is assumed; the
              routine only calculates the optimum size of the WORK array,
              returns this value as the first entry of the WORK array,
              and no error message related to LWORK is issued by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

cbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

SYNOPSIS

```
 SUBROUTINE CBDSQR( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U, LDU,
*      C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 COMPLEX VT(LDVT,*), U(LDU,*), C(LDC,*)
 INTEGER N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL D(*), E(*), WORK(*)

 SUBROUTINE CBDSQR_64( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U,
*      LDU, C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 COMPLEX VT(LDVT,*), U(LDU,*), C(LDC,*)
 INTEGER*8 N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL D(*), E(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSQR( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT, [LDVT],
*       U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:,:) :: VT, U, C
 INTEGER :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK

 SUBROUTINE BDSQR_64( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT,
*       [LDVT], U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:,:) :: VT, U, C
 INTEGER(8) :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void cbdsqr(char uplo, int n, int ncvt, int nru, int ncc, float *d, float *e, complex *vt, int ldvt, complex *u, int ldu, complex *c, int ldc, int *info);

void cbdsqr_64(char uplo, long n, long ncvt, long nru, long ncc, float *d, float *e, complex *vt, long ldvt, complex *u, long ldu, complex *c, long ldc, long *info);

PURPOSE

cbdsqr computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = Q * S * P' (P' denotes the transpose of P), where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and Q and P are orthogonal matrices.

The routine computes S, and optionally computes U * Q, P' * VT, or Q' * C, for given complex input matrices U, VT, and C.

See ``Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy," by J. Demmel and W. Kahan, LAPACK Working Note #3 (or SIAM J. Sci. Statist. Comput. vol. 11, no. 5, pp. 873-912, Sept 1990) and

``Accurate singular values and differential qd algorithms," by B. Parlett and V. Fernando, Technical Report CPAM-554, Mathematics Department, University of California at Berkeley, July 1992 for a detailed description of the algorithm.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  B is upper bidiagonal;

  = 'L':  B is lower bidiagonal.
  ```

- **N (input)**
  The order of the matrix B. N > = 0.
- **NCVT (input)**
  The number of columns of the matrix VT. NCVT > = 0.
- **NRU (input)**
  The number of rows of the matrix U. NRU > = 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B in decreasing order.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of of the bidiagonal matrix whose SVD is desired. On normal exit (INFO = 0), E is destroyed. If the algorithm does not converge (INFO > 0), D and E will contain the diagonal and superdiagonal elements of a bidiagonal matrix orthogonally equivalent to the one given as input. E(N) is used for workspace.
- **VT (input/output)**

On entry, an N-by-NCVT matrix VT. On exit, VT is overwritten by P' * VT. VT is not referenced if NCVT = 0.

- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= `max(1,N)` if NCVT > 0; LDVT >= 1 if NCVT = 0.
- **U (input/output)**
  On entry, an NRU-by-N matrix U. On exit, U is overwritten by U * Q. U is not referenced if NRU = 0.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,NRU).
- **C (input/output)**
  On entry, an N-by-NCC matrix C. On exit, C is overwritten by Q' * C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,N)` if NCC > 0; LDC >=1 if NCC = 0.
- **WORK (workspace)**
  dimension (4*N)
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  If INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm did not converge; D and E contain the
elements of a bidiagonal matrix which is orthogonally
similar to the input matrix B;  if INFO  = i, i
elements of E have not converged to zero.
```

NAME

belmm, sbelmm, dbelmm, cbelmm, zbelmm - block Ellpack format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SBELMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBELMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBELMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBELMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

```
where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in block Ellpack format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

MB            Number of block rows in matrix A

N             Number of columns in matrix C

KB            Number of block columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices
```

VAL()	scalar array of length LB*LB*BLDA*MAXBNZ containing matrix entries, stored column-major within each dense block.
BINDX()	two-dimensional integer BLDA-by-MAXBNZ array such BINDX(i,:) consists of the block column indices of the nonzero blocks in block row i, padded by the integer value i if the number of nonzero blocks is less than MAXBNZ.
BLDA	leading dimension of BINDX(:,:).
MAXBNZ	max number of nonzeros blocks per row.
LB	row and column dimension of the dense blocks composing VAL.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

belsm, sbelsm, dbelsm, cbelsm, zbelsm - block Ellpack format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SBELSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 REAL*4     ALPHA, BETA
 REAL*4     DV(MB*LB*LB), VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE DBELSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 REAL*8     ALPHA, BETA
 REAL*8     DV(MB*LB*LB), VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE CBELSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(MB*LB*LB), VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*),
*           WORK(LWORK)

 SUBROUTINE ZBELSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BLDA, MAXBNZ, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), BLDA, MAXBNZ, LB,
*           LDB, LDC, LWORK
```

```
 INTEGER*4  BINDX(BLDA,MAXBNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(MB*LB*LB), VAL(LB*LB*BLDA*MAXBNZ), B(LDB,*), C(LDC,*),
*           WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a block diagonal matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in block Ellpack format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

All blocks of A on the main diagonal MUST be triangular matrices.

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of the length MB*LB*LB consisting of the block entries of block diagonal matrix D where each block is stored in standard column-major form.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A')

2 : Hermitian (A= CONJG(A'))
3 : Triangular
4 : Skew(Anti)-Symmetric (A=-A')
5 : Diagonal
6 : Skew-Hermitian (A= -CONJG(A'))
Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-identity blocks on the main diagonal
1 : identity diagonal block
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() scalar array of length LB*LB*BLDA*MAXBNZ containing matrix entries, stored column-major within each dense block.

BINDX() two-dimensional integer BLDA-by-MAXBNZ array such BINDX(i,:) consists of the block column indices of the nonzero blocks in block row i, padded by the integer value i if the number of nonzero blocks is less than MAXBNZ. The block column indices MUST be sorted in increasing order for each block row.

BLDA leading dimension of BINDX(:,:).

MAXBNZ max number of nonzeros blocks per row.

LB row and column dimension of the dense blocks composing A.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK= -1, WORK(1) returns the minimum size of LWORK.

LWORK length of WORK array. LWORK should be at least MB*LB.

```
For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK
>=MB*LB*N_CPUS where N_CPUS is the maximum number of
processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.


If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

bscmm, sbscmm, dbscmm, cbscmm, zbscmm - block sparse column matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SBSCMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBSCMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)


 SUBROUTINE CBSCMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBSCMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 COMPLEX*16 ALPHA, BETA
```

```
COMPLEX*16 VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

where: BNNZ = BPNTRE(KB)-BPNTRB(1)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C


where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in block sparse column format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                  ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

MB            Number of block rows in matrix A

N             Number of columns in matrix C

KB            Number of block columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
```

```
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         scalar array of length LB*LB*BNNZ consisting of
              the block entries stored column-major within each
              dense block.

BINDX()       integer array of length BNNZ consisting of the
              block row indices of the block entries of A.

BPNTRB()      integer array of length KB such that
              BPNTRB(J)-BPNTRB(1)+1 points to location in BINDX
              of the first block entry of the J-th block column
              of A.

BPNTRE()      integer array of length KB such that
              BPNTRE(J)-BPNTRB(1) points to location in BINDX
              of the last block entry of the J-th block column
              of A.

LB            dimension of dense blocks composing A.

B()           rectangular array with first dimension LDB.

LDB           leading dimension of B

BETA          Scalar parameter

C()           rectangular array with first dimension LDC.

LDC           leading dimension of C

WORK()        scratch array of length LWORK. WORK is not
              referenced in the current version.

LWORK         length of WORK array. LWORK is not referenced
              in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

It is known that there exits another representation of the block sparse column format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block column in the arrays VAL and BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of block sparse column format the following calling sequence should be used

```
  CALL SBSCMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
 *            VAL, BINDX, IA, IA(2), LB,
 *            B, LDB, BETA, C, LDC, WORK, LWORK )


=cut
```

NAME

bscsm, sbscsm, dbscsm, cbscsm, zbscsm - block sparse column format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SBSCSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 REAL*4     ALPHA, BETA
 REAL*4     DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBSCSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 REAL*8     ALPHA, BETA
 REAL*8     DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBSCSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBSCSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(KB), BPNTRE(KB)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where: BNNZ = BPNTRE(KB)- BPNTRB(1)

DESCRIPTION

C <- ALPHA op(A) B + BETA C C <- ALPHA D op(A) B + BETA C
C <- ALPHA op(A) D B + BETA C

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a block diagonal matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in block sparse column format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

All blocks of A on the main diagonal MUST be triangular matrices.

=head1 ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row block scaling) 3 : Scale on right (column block scaling)
DV()	Array of the length MB*LB*LB consisting of the block entries of block diagonal matrix D where each block is stored in standard column-major form.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
  1 : lower
  2 : upper
DESCRA(3) main diagonal type
  0 : non-identity blocks on the main diagonal
  1 : identity diagonal block
DESCRA(4) Array base (NOT IMPLEMENTED)
  0 : C/C++ compatible
  1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
  0 : unknown
  1 : no repeated indices

VAL() scalar array of length LB*LB*BNNZ consisting of the block entries stored column-major within each dense block.

BINDX() integer array of length BNNZ consisting of the block row indices of the block entries of A. The block row indices MUST be sorted in increasing order for each block column.

BPNTRB() integer array of length KB such that BPNTRB(J)-BPNTRB(1)+1 points to location in BINDX of the first block entry of the J-th block column of A.

BPNTRE() integer array of length KB such that BPNTRE(J)-BPNTRB(1) points to location in BINDX of the last block entry of the J-th block column of A.

LB dimension of dense blocks composing A.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK= -1, WORK(1) returns the optimum size of LWORK.

LWORK length of WORK array. LWORK should be at least MB*LB.

For good performance, LWORK should generally be larger.

```
For optimum performance on multiple processors, LWORK
>=MB*LB*N_CPUS where N_CPUS is the maximum number of
processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.


If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

1. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

2. It is known that there exits another representation of the block sparse column format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block column in the arrays VAL and BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of block sparse column format the following calling sequence should be used

```
 CALL SBSCSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, IA, IA(2), LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
```

NAME

bsrmm, sbsrmm, dbsrmm, cbsrmm, zbsrmm - block sparse row format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SBSRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBSRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBSRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBSRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where: BNNZ = BPNTRE(MB)-BPNTRB(1)

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in block sparse row format and
op(A) is one of

op(A) = A   or   op(A) = A'   or   op(A) = conjg(A').
(' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix A is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
KB	Number of block columns in matrix A
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A')) DESCRA(2) upper/lower triangular indicator 1 : lower 2 : upper DESCRA(3) main diagonal type 0 : non-unit 1 : unit DESCRA(4) Array base (NOT IMPLEMENTED) 0 : C/C++ compatible

```
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices


VAL()         scalar array of length LB*LB*BNNZ consisting
              of the block  entries stored column-major within
              each dense block.

BINDX()       integer array of length BNNZ consisting of the
              block column indices of the block entries of A.

BPNTRB()      integer array of length MB such that
              BPNTRB(J)-BPNTRB(1)+1 points to location in BINDX
              of the first block entry of the J-th block row of A.

BPNTRE()      integer array of length MB such that
              BPNTRE(J)-BPNTRB(1) points to location in BINDX
              of the last block entry of the J-th block row of A.

LB            dimension of dense blocks composing A.

B()           rectangular array with first dimension LDB.

LDB           leading dimension of B

BETA          Scalar parameter

C()           rectangular array with first dimension LDC.

LDC           leading dimension of C

WORK()        scratch array of length LWORK. WORK is not
              referenced in the current version.


LWORK         length of WORK array. LWORK is not referenced
              in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

It is known that there exits another representation of the block sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block row in the arrays VAL and BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of block sparse row format the following calling sequence should be used

```
  CALL SBSRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
 *            VAL, BINDX, IA, IA(2), LB,
 *            B, LDB, BETA, C, LDC, WORK, LWORK )


=cut
```

NAME

bsrsm, sbsrsm, dbsrsm, cbsrsm, zbsrsm - block sparse row format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SBSRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 REAL*4     ALPHA, BETA
 REAL*4     DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DBSRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 REAL*8     ALPHA, BETA
 REAL*8     DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CBSRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZBSRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, BPNTRB, BPNTRE, LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LB,
*           LDB, LDC, LWORK
 INTEGER*4  BINDX(BNNZ), BPNTRB(MB), BPNTRE(MB)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(MB*LB*LB), VAL(LB*LB*BNNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where: BNNZ = BPNTRE(MB)-BPNTRB(1)

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a block diagonal matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in block sparse row format format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

All blocks of A on the main diagonal MUST be triangular matrices.

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row block scaling) 3 : Scale on right (column block scaling)
DV()	Array of the length MB*LB*LB consisting of the block entries of block diagonal matrix D where each block is stored in standard column-major form.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular

4 : Skew(Anti)-Symmetric (A=-A')
5 : Diagonal
6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-identity blocks on the main diagonal
1 : identity diagonal block
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() scalar array of length LB*LB*BNNZ consisting of the block entries stored column-major within each dense block.

BINDX() integer array of length BNNZ consisting of the block column indices of the block entries of A. The block column indices MUST be sorted in increasing order for each block row.

BPNTRB() integer array of length MB such that BPNTRB(J)-BPNTRB(1)+1 points to location in BINDX of the first block entry of the J-th block row of A.

BPNTRE() integer array of length MB such that BPNTRE(J)-BPNTRB(1) points to location in BINDX of the last block entry of the J-th block row of A.

LB dimension of dense blocks composing A.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK= -1, WORK(1) returns the optimum size of LWORK.

LWORK length of WORK array. LWORK should be at least

```
MB*LB.


For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK
>=MB*LB*N_CPUS where N_CPUS is the maximum number of
processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.


If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

1. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

2. It is known that there exits another representation of the block sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block row in the arrays VAL and BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of block sparse row format the following calling sequence should be used

```
 CALL SBSRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, BINDX, IA, IA(2), LB,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
```

NAME

ccnvcor - compute the convolution or correlation of complex vectors

---

SYNOPSIS

```
 SUBROUTINE CCNVCOR( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M, Y,
*      IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 COMPLEX X(*), Y(*), Z(*), WORK(*)
 INTEGER NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK

 SUBROUTINE CCNVCOR_64( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M,
*      Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 COMPLEX X(*), Y(*), Z(*), WORK(*)
 INTEGER*8 NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR( CNVCOR, FOUR, NX, X, IFX, [INCX], NY, NPRE, M, Y,
*       IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 COMPLEX, DIMENSION(:) :: X, Y, Z, WORK
 INTEGER :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK

 SUBROUTINE CNVCOR_64( CNVCOR, FOUR, NX, X, IFX, [INCX], NY, NPRE, M,
*       Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 COMPLEX, DIMENSION(:) :: X, Y, Z, WORK
 INTEGER(8) :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
```

C INTERFACE

#include <sunperf.h>

void ccnvcor(char cnvcor, char four, int nx, complex *x, int ifx, int incx, int ny, int npre, int m, complex *y, int ify, int inc1y, int inc2y, int nz, int k, complex *z, int ifz, int inc1z, int inc2z, complex *work, int lwork);

void ccnvcor_64(char cnvcor, char four, long nx, complex *x, long ifx, long incx, long ny, long npre, long m, complex *y, long ify, long inc1y, long inc2y, long nz, long k, complex *z, long ifz, long inc1z, long inc2z, complex *work, long lwork);

PURPOSE

ccnvcor computes the convolution or correlation of complex vectors.

ARGUMENTS

- **CNVCOR (input)**

  ```
  CHARACTER
  ```

  \'V' or 'v' if convolution is desired, 'R' or 'r' if correlation is desired.

- **FOUR (input)**

  ```
  CHARACTER
  ```

  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' if the computation should be done directly from the definition. The Fourier transform method is generally faster, but it may introduce noticeable errors into certain results, notably when both the real and imaginary parts of the filter and data vectors consist entirely of integers or vectors where elements of either the filter vector or a given data vector differ significantly in magnitude from the 1-norm of the vector.

- **NX (input)**
  Length of the filter vector. NX > = 0. CCNVCOR will return immediately if NX = 0.

- **X (input)**

  ```
  dimension(*)
  ```

  Filter vector.

- **IFX (input)**
  Index of the first element of X. NX > = IFX > = 1.

- **INCX (input)**
  Stride between elements of the filter vector in X. INCX > 0.

- **NY (input)**
  Length of the input vectors. NY > = 0. CCNVCOR will return immediately if NY = 0.

- **NPRE (input)**
  The number of implicit zeros prepended to the Y vectors. NPRE > = 0.

- **M (input)**
  Number of input vectors. M > = 0. CCNVCOR will return immediately if M = 0.

- **Y (input)**

  ```
  dimension(*)
  ```

  Input vectors.

- **IFY (input)**
  Index of the first element of Y. NY > = IFY > = 1.

- **INC1Y (input)**
  Stride between elements of the input vectors in Y. INC1Y > 0.

- **INC2Y (input)**
  Stride between the input vectors in Y. INC2Y > 0.

- **NZ (input)**
  Length of the output vectors. NZ > = 0. CCNVCOR will return immediately if NZ = 0. See the Notes section below for information about how this argument interacts with NX and NY to control circular versus end-off shifting.

- **K (input)**
  Number of Z vectors. K > = 0. If K = 0 then CCNVCOR will return immediately. If K < M then only the first K input vectors will be processed. If K > M then M input vectors will be processed.

- **Z (output)**

`dimension(*)`

Result vectors.

- **IFZ (input)**
  Index of the first element of Z. NZ > = IFZ > = 1.
- **INC1Z (input)**
  Stride between elements of the output vectors in Z. INC1Z > 0.
- **INC2Z (input)**
  Stride between the output vectors in Z. INC2Z > 0.
- **WORK (input/output)**
  (input/scratch) `dimension(LWORK)`

  Scratch space. Before the first call to CCNVCOR with particular values of the integer arguments the first element of WORK must be set to zero. If WORK is written between calls to CCNVCOR or if CCNVCOR is called with different values of the integer arguments then the first element of WORK must again be set to zero before each call. If WORK has not been written and the same values of the integer arguments are used then the first element of WORK to zero. This can avoid certain initializations that store their results into WORK, and avoiding the initialization can make CCNVCOR run faster.
- **LWORK (input)**
  Length of WORK. LWORK > = 2*MAX(NX,NY,NZ)+8.

NAME

ccnvcor2 - compute the convolution or correlation of complex matrices

---

SYNOPSIS

```
 SUBROUTINE CCNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*      SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*      LDZ, WORK, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX X(LDX,*), Y(LDY,*), Z(LDZ,*), WORK(*)
 INTEGER MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK

 SUBROUTINE CCNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*      SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*      LDZ, WORK, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX X(LDX,*), Y(LDY,*), Z(LDZ,*), WORK(*)
 INTEGER*8 MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: X, Y, Z
 INTEGER :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK

 SUBROUTINE CNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: X, Y, Z
 INTEGER(8) :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
```

C INTERFACE

#include <sunperf.h>

void ccnvcor2(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, int mx, int nx, complex *x, int ldx, int my, int ny, int mpre, int npre, complex *y, int ldy, int mz, int nz, complex *z, int ldz, complex *work, int lwork);

void ccnvcor2_64(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, long mx, long nx, complex *x, long ldx, long my, long ny, long mpre, long npre, complex *y, long ldy, long mz, long nz, complex *z, long ldz, complex *work, long lwork);

PURPOSE

ccnvcor2 computes the convolution or correlation of complex matrices.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' to compute convolution, 'R' or 'r' to compute correlation.
- **METHOD (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' to compute directly from the definition.
- **TRANSX (input)**
  \'N' or 'n' if X is the filter matrix, 'T' or 't' if `transpose(X)` is the filter matrix.
- **SCRATCHX (input)**
  \'N' or 'n' if X must be preserved, 'S' or 's' if X can be used as scratch space. The contents of X are undefined after returning from a call in which X is allowed to be used for scratch.
- **TRANSY (input)**
  \'N' or 'n' if Y is the input matrix, 'T' or 't' if `transpose(Y)` is the input matrix.
- **SCRATCHY (input)**
  \'N' or 'n' if Y must be preserved, 'S' or 's' if Y can be used as scratch space. The contents of Y are undefined after returning from a call in which Y is allowed to be used for scratch.
- **MX (input)**
  Number of rows in the filter matrix. MX > = 0.
- **NX (input)**
  Number of columns in the filter matrix. NX > = 0.
- **X (input)**
  On entry, the filter matrix. Unchanged on exit if SCRATCHX is 'N' or 'n', undefined on exit if SCRATCHX is 'S' or 's'.
- **LDX (input)**
  Leading dimension of the array that contains the filter matrix.
- **MY (input)**
  Number of rows in the input matrix. MY > = 0.
- **NY (input)**
  Number of columns in the input matrix. NY > = 0.
- **MPRE (input)**
  Number of implicit zeros to prepend to each row of the input matrix. MPRE > = 0.

- **NPRE (input)**
  Number of implicit zeros to prepend to each column of the input matrix. NPRE > = 0.
- **Y (input)**
  Input matrix. Unchanged on exit if SCRATCHY is 'N' or 'n', undefined on exit if SCRATCHY is 'S' or 's'.
- **LDY (input)**
  Leading dimension of the array that contains the input matrix.
- **MZ (input)**
  Number of rows in the output matrix. MZ > = 0. CCNVCOR2 will return immediately if MZ = 0.
- **NZ (input)**
  Number of columns in the output matrix. NZ > = 0. CCNVCOR2 will return immediately if NZ = 0.
- **Z (output)**

  ```
  dimension(LDZ,*)
  ```

  Result matrix.
- **LDZ (input)**
  Leading dimension of the array that contains the result matrix. LDZ > = MAX(1,MZ).
- **WORK (input/output)**
  (input/scratch) `dimension(LWORK)`

  On entry for the first call to CCNVCOR2, `WORK(1)` must contain CMPLX(0.0,0.0). After the first call, `WORK(1)` must be set to `CMPLX(0.0,0.0)` iff WORK has been altered since the last call to this subroutine or if the sizes of the arrays have changed.
- **LWORK (input)**
  Length of the work vector. If the FFT is to be used then for best performance LWORK should be at least 30 words longer than the amount of memory needed to hold the trig tables. If the FFT is not used, the value of LWORK is unimportant.

NAME

coomm, scoomm, dcoomm, ccoomm, zcoomm - coordinate matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SCOOMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, JNDX, NNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), NNZ
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), JNDX(NNZ)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DCOOMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, JNDX, NNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), NNZ
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), JNDX(NNZ)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)


 SUBROUTINE CCOOMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, JNDX, NNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), NNZ
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), JNDX(NNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZCOOMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, JNDX, NNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), NNZ
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), JNDX(NNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in coordinate format and
op(A) is one of

op(A) = A   or   op(A) = A'   or   op(A) = conjg(A').
(' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
K	Number of columns in matrix A
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A')) DESCRA(2) upper/lower triangular indicator 1 : lower 2 : upper DESCRA(3) main diagonal type 0 : non-unit 1 : unit DESCRA(4) Array base (NOT IMPLEMENTED) 0 : C/C++ compatible 1 : Fortran compatible DESCRA(5) repeated indices? (NOT IMPLEMENTED) 0 : unknown

1 : no repeated indices

VAL()	scalar array of length NNZ consisting of the non-zero entries of A, in any order.
INDX()	integer array of length NNZ consisting of the corresponding row indices of the entries of A.
JNDX()	integer array of length NNZ consisting of the corresponding column indices of the entries of A.
NNZ	number of non-zero elements in A.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

ccopy - Copy x to y

SYNOPSIS

```
SUBROUTINE CCOPY( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE CCOPY_64( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE COPY( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE COPY_64( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void ccopy(int n, complex *x, int incx, complex *y, int incy);

void ccopy_64(long n, complex *x, long incx, complex *y, long incy);

PURPOSE

ccopy Copy x to y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  of DIMENSION at least (1 + (m - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cscmm, scscmm, dcscmm, ccscmm, zcscmm - compressed sparse column format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SCSCMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DCSCMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CCSCMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZCSCMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

```
  where NNZ = PNTRE(K)-PNTRB(1)
```

DESCRIPTION

```
          C <- alpha op(A) B + beta C


 where ALPHA and BETA are scalar, C and B are dense matrices,
 A is a matrix represented in compressed sparse column format and
 op( A )  is one  of

 op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)


=head1 ARGUMENTS

 TRANSA         Indicates how to operate with the sparse matrix
                  0 : operate with matrix
                  1 : operate with transpose matrix
                  2 : operate with the conjugate transpose of matrix.
                      2 is equivalent to 1 if matrix is real.

 M              Number of rows in matrix A

 N              Number of columns in matrix C

 K              Number of columns in matrix A

 ALPHA          Scalar parameter

 DESCRA()       Descriptor argument.  Five element integer array
                DESCRA(1) matrix structure
                  0 : general
                  1 : symmetric (A=A')
                  2 : Hermitian (A= CONJG(A'))
                  3 : Triangular
                  4 : Skew(Anti)-Symmetric (A=-A')
                  5 : Diagonal
                  6 : Skew-Hermitian (A= -CONJG(A'))
                DESCRA(2) upper/lower triangular indicator
                  1 : lower
                  2 : upper
                DESCRA(3) main diagonal type
                  0 : non-unit
                  1 : unit
                DESCRA(4) Array base  (NOT IMPLEMENTED)
                  0 : C/C++ compatible
                  1 : Fortran compatible
                DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                  0 : unknown
                  1 : no repeated indices
```

VAL()	scalar array of length NNZ consisting of nonzero entries of A.
INDX()	integer array of length NNZ consisting of the row indices of nonzero entries of A.
PNTRB()	integer array of length K such that PNTRB(J)-PNTRB(1)+1 points to location in VAL of the first nonzero element in column J.
PNTRE()	integer array of length K such that PNTRE(J)-PNTRB(1) points to location in VAL of the last nonzero element in column J.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

It is known that there exits another representation of the compressed sparse column format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems'', WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each column in the arrays VAL and INDX is used instead of two arrays PNTRB and PNTRE. To use the routine with this kind of sparse column format the following calling sequence should be used SUBROUTINE SCSCMM(TRANSA, M, N, K, ALPHA, DESCRA, * VAL, INDX, IA, IA(2), B, LDB, BETA, * C, LDC, WORK, LWORK)

NAME

cscsm, scscsm, dcscsm, ccscsm, zcscsm - compressed sparse column format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SCSCSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DCSCSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CCSCSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZCSCSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(K), PNTRE(K)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where NNZ = PNTRE(K)-PNTRB(1)

DESCRIPTION

C <- ALPHA op(A) B + BETA C C <- ALPHA D op(A) B + BETA C
C <- ALPHA op(A) D B + BETA C

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a diagonal scaling matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in compressed sparse column format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length M containing the diagonal entries of the scaling diagonal matrix D.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
  1 : lower
  2 : upper
DESCRA(3) main diagonal type
  0 : non-unit
  1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
  0 : C/C++ compatible
  1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
  0 : unknown
  1 : no repeated indices

VAL() scalar array of length NNZ consisting of nonzero entries of A.

INDX() integer array of length NNZ consisting of the row indices of nonzero entries of A. (Row indices MUST be sorted in increasing order for each column).

PNTRB() integer array of length K such that PNTRB(J)-PNTRB(1)+1 points to location in VAL of the first nonzero element in column J.

PNTRE() integer array of length K such that PNTRE(J)-PNTRB(1) points to location in VAL of the last nonzero element in column J.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.

On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least M.

For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program.

```
If LWORK=0, the routine is to allocate workspace needed.


If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum  size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

1. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

2. It is known that there exits another representation of the compressed sparse column format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems'', WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each column in the arrays VAL and INDX is used instead of two arrays PNTRB and PNTRE. To use the routine with this kind of sparse column format the following calling sequence should be used SUBROUTINE SCSCSM(TRANSA, M, N, UNITD, DV, ALPHA, DESCRA, * VAL, INDX, IA, IA(2), B, LDB, BETA, * C, LDC, WORK, LWORK)

NAME

csrmm, scsrmm, dcsrmm, ccsrmm, zcsrmm - compressed sparse row format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SCSRMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DCSRMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CCSRMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZCSRMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where NNZ = PNTRE(M)-PNTRB(1)

DESCRIPTION

C <- alpha op(A) B + beta C

where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in compressed sparse row format and
op(A) is one of

op(A) = A or op(A) = A' or op(A) = conjg(A').
(' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
K	Number of columns in matrix A
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A')) DESCRA(2) upper/lower triangular indicator 1 : lower 2 : upper DESCRA(3) main diagonal type 0 : non-unit 1 : unit DESCRA(4) Array base (NOT IMPLEMENTED) 0 : C/C++ compatible

```
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         scalar array of length NNZ consisting of nonzero
              entries of A.

INDX()        integer array of length NNZ consisting of the
              column indices of nonzero entries of A.

PNTRB()       integer array of length M such that PNTRB(J)-PNTRB(1)+1
              points to location in VAL of the first nonzero element
              in row J.

PNTRE()       integer array of length M such that PNTRE(J)-PNTRB(1)
              points to location in VAL of the last nonzero element
              in row J.

B()           rectangular array with first dimension LDB.

LDB           leading dimension of B

BETA          Scalar parameter

C()           rectangular array with first dimension LDC.

LDC           leading dimension of C

WORK()        scratch array of length LWORK. WORK is not
              referenced in the current version.


LWORK         length of WORK array. LWORK is not referenced
              in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

It is known that there exits another representation of the compressed sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each row in the arrays VAL and INDX is used instead of two arrays PNTRB and PNTRE. To use the routine with this kind of compressed sparse row format the following calling sequence should be used SUBROUTINE SCSRMM(TRANSA, M, N, K, ALPHA, DESCRA, * VAL, INDX, IA, IA(2), B, LDB, BETA, * C, LDC, WORK, LWORK)

NAME

csrsm, scsrsm, dcsrsm, ccsrsm, zcsrsm - compressed sparse row format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SCSRSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DCSRSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CCSRSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZCSRSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTRB, PNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTRB(M), PNTRE(M)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

where NNZ = PNTRE(M)-PNTRB(1)

DESCRIPTION

C <- ALPHA op(A) B + BETA C C <- ALPHA D op(A) B + BETA C
C <- ALPHA op(A) D B + BETA C

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a diagonal scaling matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in compressed sparse row format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length M containing the diagonal entries of the scaling diagonal matrix D.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, only DESCRA(1)=3 is supported.

DESCRA(2) upper/lower triangular indicator
  1 : lower
  2 : upper
DESCRA(3) main diagonal type
  0 : non-unit
  1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
  0 : C/C++ compatible
  1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
  0 : unknown
  1 : no repeated indices

VAL() scalar array of length NNZ consisting of nonzero entries of A.

INDX() integer array of length NNZ consisting of the column indices of nonzero entries of A (column indices MUST be sorted in increasing order for each row)

PNTRB() integer array of length M such that PNTRB(J)-PNTRB(1)+1 points to location in VAL of the first nonzero element in row J.

PNTRE() integer array of length M such that PNTRE(J)-PNTRB(1) points to location in VAL of the last nonzero element in row J.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.

On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least M.

For good performance, LWORK should generally be larger. For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.

```
If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

1. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

2. It is known that there exits another representation of the compressed sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of three array instead of the four used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each row in the arrays VAL and INDX is used instead of two arrays PNTRB and PNTRE. To use the routine with this kind of compressed sparse row format the following calling sequence should be used SUBROUTINE SCSRSM(TRANSA, M, N, UNITD, DV, ALPHA, DESCRA, * VAL, INDX, IA, IA(2), B, LDB, BETA, C, * LDC, WORK, LWORK)

NAME

diamm, sdiamm, ddiamm, cdiamm, zdiamm - diagonal format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SDIAMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DDIAMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CDIAMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZDIAMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

```
where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in diagonal format and op( A ) is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                  ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

M             Number of rows in matrix A

N             Number of columns in matrix C

K             Number of columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         two-dimensional LDA-by-NDIAG array such that VAL(:,I)
```

	consists of non-zero elements on diagonal IDIAG(I) of A. Diagonals in the lower triangular part of A are padded from the top, and those in the upper triangular part are padded from the bottom.
LDA	leading dimension of VAL, must be .GE. MIN(M,K)
IDIAG()	integer array of length NDIAG consisting of the corresponding diagonal offsets of the non-zero diagonals of A in VAL. Lower triangular diagonals have negative offsets, the main diagonal has offset 0, and upper triangular diagonals have positive offset.
NDIAG	number of non-zero diagonals in A.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

diasm, sdiasm, ddiasm, cdiasm, zdiasm - diagonal format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SDIASM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DDIASM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CDIASM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZDIASM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, LDA, IDIAG, NDIAG,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, NDIAG,
*           LDB, LDC, LWORK
 INTEGER*4  IDIAG(NDIAG)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(LDA,NDIAG), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices,
D is a diagonal scaling matrix, A is a unit, or non-unit, upper or
lower triangular matrix represented in diagonal format and
op(A) is one of

```
 op( A ) = inv(A) or  op( A ) = inv(A')  or  op( A ) =inv(conjg( A' ))
```

(inv denotes matrix inverse, ' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length M containing the diagonal entries of the scaling diagonal matrix D.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, only DESCRA(1)=3 is supported.

DESCRA(2) upper/lower triangular indicator
  1 : lower
  2 : upper
DESCRA(3) main diagonal type
  0 : non-unit
  1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
  0 : C/C++ compatible
  1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
  0 : unknown
  1 : no repeated indices

VAL() two-dimensional LDA-by-NDIAG array such that VAL(:,I) consists of non-zero elements on diagonal IDIAG(I) of A. Diagonals in the lower triangular part of A are padded from the top, and those in the upper triangular part are padded from the bottom.

LDA leading dimension of VAL, must be .GE. MIN(M,K)

IDIAG() integer array of length NDIAG consisting of the corresponding diagonal offsets of the non-zero diagonals of A in VAL. Lower triangular diagonals have negative offsets, the main diagonal has offset 0, and upper triangular diagonals have positive offset. Elements of IDIAG of MUST be sorted in increasing order.

NDIAG number of non-zero diagonals in A.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least M.

For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.

```
If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

cdotc - compute the dot product of two vectors `conjg(x)` and y.

SYNOPSIS

```
COMPLEX FUNCTION CDOTC( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

COMPLEX FUNCTION CDOTC_64( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
COMPLEX FUNCTION DOTC( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

COMPLEX FUNCTION DOTC_64( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

complex cdotc(int n, complex *x, int incx, complex *y, int incy);

complex cdotc_64(long n, complex *x, long incx, complex *y, long incy);

PURPOSE

cdotc compute the dot product of `conjg(x)` and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cdotci - Compute the complex conjugated indexed dot product.

SYNOPSIS

```
COMPLEX FUNCTION CDOTCI(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

COMPLEX FUNCTION CDOTCI_64(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE COMPLEX FUNCTION DOTCI([NZ], X, INDX, Y)

```
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

COMPLEX FUNCTION DOTCI_64([NZ], X, INDX, Y)

COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CDOTCI Compute the complex conjugated indexed dot product of a complex sparse vector x stored in compressed form with a complex vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + conjg(x(i)) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

cdotu - compute the dot product of two vectors x and y.

SYNOPSIS

```
COMPLEX FUNCTION CDOTU( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

COMPLEX FUNCTION CDOTU_64( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
COMPLEX FUNCTION DOT( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

COMPLEX FUNCTION DOT_64( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

complex cdotu(int n, complex *x, int incx, complex *y, int incy);

complex cdotu_64(long n, complex *x, long incx, complex *y, long incy);

PURPOSE

cdotu compute the dot product of x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cdotui - Compute the complex unconjugated indexed dot product.

SYNOPSIS

```
COMPLEX FUNCTION CDOTCI(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

COMPLEX FUNCTION CDOTCI_64(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE COMPLEX FUNCTION DOTCI([NZ], X, INDX, Y)

```
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

COMPLEX FUNCTION DOTCI_64([NZ], X, INDX, Y)

COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CDOTUI Compute the complex unconjugated indexed dot product of a complex sparse vector x stored in compressed form with a complex vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + x(i) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

ellmm, sellmm, dellmm, cellmm, zellmm - Ellpack format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SELLMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DELLMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CELLMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZELLMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in  Ellpack format format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

M             Number of rows in matrix A

N             Number of columns in matrix C

K             Number of columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         two-dimensional LDA-by-MAXNZ array such that VAL(I,:)
              consists of non-zero elements in row I of A, padded by
```

	zero values if the row contains less than MAXNZ.
INDX()	two-dimensional integer BLDA-by-MAXBNZ array such INDX(I,:) consists of the column indices of the nonzero elements in row I, padded by the integer value I if the number of nonzeros is less than MAXNZ.
LDA	leading dimension of VAL and INDX.
MAXNZ	max number of nonzeros elements per row.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

ellsm, sellsm, dellsm, cellsm, zellsm - Ellpack format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SELLSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DELLSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CELLSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DELLSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, LDA, MAXNZ,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), LDA, MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(LDA,MAXNZ)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(LDA,MAXNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices,
D is a diagonal scaling matrix, A is a unit, or non-unit, upper or
lower triangular matrix represented in Ellpack format and
op(A) is one of

```
 op( A ) = inv(A) or  op( A ) = inv(A')  or  op( A ) =inv(conjg( A' ))
 (inv denotes matrix inverse,  ' indicates matrix transpose)
```

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length M containing the diagonal entries of the scaling diagonal matrix D.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A')) Note: For the routine, only DESCRA(1)=3 is supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-unit
1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() two-dimensional LDA-by-MAXNZ array such that VAL(I,:) consists of non-zero elements in row I of A, padded by zero values if the row contains less than MAXNZ.

INDX() two-dimensional integer LDA-by-MAXNZ array such INDX(I,:) consists of the column indices of the nonzero elements in row I, padded by the integer value I if the number of nonzeros is less than MAXNZ. The column indices MUST be sorted in increasing order for each row.

LDA leading dimension of VAL and INDX.

MAXNZ max number of nonzeros elements per row.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.

On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least M.

For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.

```
If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

cfft2b - compute a periodic sequence from its Fourier coefficients. The xFFT operations are unnormalized, so a call of xFFT2F followed by a call of xFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
SUBROUTINE CFFT2B( M, N, A, LDA, WORK, LWORK)
COMPLEX A(LDA,*)
INTEGER M, N, LDA, LWORK
REAL WORK(*)

SUBROUTINE CFFT2B_64( M, N, A, LDA, WORK, LWORK)
COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, LWORK
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2B( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT2B_64( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft2b(int m, int n, complex *a, int lda, float *work, int lwork);

void cfft2b_64(long m, long n, complex *a, long lda, float *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **A (input/output)**
  On entry, a two-dimensional array `A(M,N)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **WORK (input)**
  On input, workspace WORK must have been initialized by CFFT2I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4 * (M + N) + 30)

NAME

cfft2f - compute the Fourier coefficients of a periodic sequence. The xFFT operations are unnormalized, so a call of xFFT2F followed by a call of xFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
SUBROUTINE CFFT2F( M, N, A, LDA, WORK, LWORK)
COMPLEX A(LDA,*)
INTEGER M, N, LDA, LWORK
REAL WORK(*)

SUBROUTINE CFFT2F_64( M, N, A, LDA, WORK, LWORK)
COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, LWORK
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2F( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT2F_64( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft2f(int m, int n, complex *a, int lda, float *work, int lwork);

void cfft2f_64(long m, long n, complex *a, long lda, float *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **A (input/output)**
  On entry, a two-dimensional array `A(M,N)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **WORK (input)**
  On input, workspace WORK must have been initialized by CFFT2I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4 * (M + N) + 30)

NAME

cfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

SYNOPSIS

```
SUBROUTINE CFFT2I( M, N, WORK)
INTEGER M, N
REAL WORK(*)

SUBROUTINE CFFT2I_64( M, N, WORK)
INTEGER*8 M, N
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE CFFT2I( M, N, WORK)
INTEGER :: M, N
REAL, DIMENSION(:) :: WORK

SUBROUTINE CFFT2I_64( M, N, WORK)
INTEGER(8) :: M, N
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft2i(int m, int n, float *work);

void cfft2i_64(long m, long n, float *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (4 * (M + N) + 30) or greater. CFFT2I needs to be called only once to initialize array WORK before calling CFFT2F and/or CFFT2B if M, N and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

cfft3b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of CFFT3F followed by a call of CFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
SUBROUTINE CFFT3B( M, N, K, A, LDA, LD2A, WORK, LWORK)
COMPLEX A(LDA,LD2A,*)
INTEGER M, N, K, LDA, LD2A, LWORK
REAL WORK(*)

SUBROUTINE CFFT3B_64( M, N, K, A, LDA, LD2A, WORK, LWORK)
COMPLEX A(LDA,LD2A,*)
INTEGER*8 M, N, K, LDA, LD2A, LWORK
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3B( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX, DIMENSION(:,:,:) :: A
INTEGER :: M, N, K, LDA, LD2A, LWORK
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT3B_64( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX, DIMENSION(:,:,:) :: A
INTEGER(8) :: M, N, K, LDA, LD2A, LWORK
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft3b(int m, int n, int k, complex *a, int lda, int ld2a, float *work, int lwork);

void cfft3b_64(long m, long n, long k, complex *a, long lda, long ld2a, float *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **K (input)**
  Number of planes to be transformed. These subroutines are most efficient when K is a product of small primes. K >= 0.
- **A (input/output)**
  On entry, a three-dimensional array `A(LDA,LD2A,K)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **LD2A (input)**
  Second dimension of the array containing the data to be transformed. LD2A >= N.
- **WORK (input)**
  On input, workspace WORK must have been initialized by CFFT3I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4*(M + N + K) + 45).

NAME

cfft3f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of CFFT3F followed by a call of CFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
SUBROUTINE CFFT3F( M, N, K, A, LDA, LD2A, WORK, LWORK)
COMPLEX A(LDA,LD2A,*)
INTEGER M, N, K, LDA, LD2A, LWORK
REAL WORK(*)

SUBROUTINE CFFT3F_64( M, N, K, A, LDA, LD2A, WORK, LWORK)
COMPLEX A(LDA,LD2A,*)
INTEGER*8 M, N, K, LDA, LD2A, LWORK
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3F( [M], [N], [K], A, [LDA], [LD2A], WORK, LWORK)
COMPLEX, DIMENSION(:,:,:) :: A
INTEGER :: M, N, K, LDA, LD2A, LWORK
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT3F_64( [M], [N], [K], A, [LDA], [LD2A], WORK, LWORK)
COMPLEX, DIMENSION(:,:,:) :: A
INTEGER(8) :: M, N, K, LDA, LD2A, LWORK
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft3f(int m, int n, int k, complex *a, int lda, int ld2a, float *work, int lwork);

void cfft3f_64(long m, long n, long k, complex *a, long lda, long ld2a, float *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **K (input)**
  Number of planes to be transformed. These subroutines are most efficient when K is a product of small primes. K >= 0.
- **A (input/output)**
  On entry, a three-dimensional array `A(M,N,K)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **LD2A (input)**
  Second dimension of the array containing the data to be transformed. LD2A >= N.
- **WORK (input)**
  On input, workspace WORK must have been initialized by CFFT3I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4*(M + N + K) + 45).

NAME

cfft3i - initialize the array WSAVE, which is used in both CFFT3F and CFFT3B.

SYNOPSIS

```
SUBROUTINE CFFT3I( M, N, K, WORK)
INTEGER M, N, K
REAL WORK(*)

SUBROUTINE CFFT3I_64( M, N, K, WORK)
INTEGER*8 M, N, K
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE CFFT3I( M, N, K, WORK)
INTEGER :: M, N, K
REAL, DIMENSION(:) :: WORK

SUBROUTINE CFFT3I_64( M, N, K, WORK)
INTEGER(8) :: M, N, K
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void cfft3i(int m, int n, int k, float *work);

void cfft3i_64(long m, long n, long k, float *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **K (input)**
  Number of planes to be transformed. K > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (4*(M + N + K) + 45) or greater. CFFT3I needs to be called only once to initialize array WORK before calling CFFT3F and/or CFFT3B if M, N, K and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

cfftb - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of CFFTF followed by a call of CFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE CFFTB( N, X, WSAVE)
COMPLEX X(*)
INTEGER N
REAL WSAVE(*)

SUBROUTINE CFFTB_64( N, X, WSAVE)
COMPLEX X(*)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [N], X, WSAVE)
COMPLEX, DIMENSION(:) :: X
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE FFTB_64( [N], X, WSAVE)
COMPLEX, DIMENSION(:) :: X
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void cfftb(int n, complex *x, float *wsave);

void cfftb_64(long n, complex *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (4 * N + 15) or greater and must have been initialized by CFFTI.

NAME

cfftc - initialize the trigonometric weight and factor tables or compute the Fast Fourier transform (forward or inverse) of a complex sequence. =head1 SYNOPSIS

```
 SUBROUTINE CFFTC( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX X(*), Y(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)

 SUBROUTINE CFFTC_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX X(*), Y(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX, DIMENSION(:) :: X, Y
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK,
*       [LWORK], IERR)
 COMPLEX, DIMENSION(:) :: X, Y
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void cfftc(int iopt, int n, float scale, complex *x, complex *y, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cfftc_64(long iopt, long n, float scale, complex *x, complex *y, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cfftc initializes the trigonometric weight and factor tables or computes the Fast Fourier transform (forward or inverse) of a complex sequence as follows: .Ve

```
                N-1
```

Y(k) = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

k ranges from 0 to N-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  On entry, X is a complex array of dimension at least N that contains the sequence to be transformed.
- **Y (output)**

Complex array of dimension at least N that contains the transform results. X and Y may be the same array starting at the same memory location. Otherwise, it is assumed that there is no overlap between X and Y in memory.

- **TRIGS (input/output)**
  Real array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least 2*N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than 2*N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

cfftc2 - initialize the trigonometric weight and factor tables or compute the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTC2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)

 SUBROUTINE CFFTC2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X, Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT2_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X, Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void cfftc2(int iopt, int n1, int n2, float scale, complex *x, int ldx, complex *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cfftc2_64(long iopt, long n1, long n2, float scale, complex *x, long ldx, complex *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cfftc2 initializes the trigonometric weight and factor tables or computes the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional complex array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the columns of the input array. One-dimensional FFTs are then computed along the rows of the intermediate results. .Ve

```
                    N2-1  N1-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**

Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX, N2) that contains input data to be transformed.
- **LDX (input)**
  Leading dimension of X. LDX > = N1 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = LDX Else LDY > = N1 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least 2*MAX(N1,N2)*NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1)

  -5 = (LDY < N1) or (LDY not equal LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < 2*MAX(N1,N2)*NCPUS)

  -7 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, entire output array Y(1:LDY, 1:N2) is overwritten.

NAME

cfftc3 - initialize the trigonometric weight and factor tables or compute the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTC3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX1,LDX2,*), Y(LDY1,LDY2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)

 SUBROUTINE CFFTC3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX1,LDX2,*), Y(LDY1,LDY2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: X, Y
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT3_64( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1],
*       LDX2, Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: X, Y
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void cfftc3(int iopt, int n1, int n2, int n3, float scale, complex *x, int ldx1, int ldx2, complex *y, int ldy1, int ldy2, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cfftc3_64(long iopt, long n1, long n2, long n3, float scale, complex *x, long ldx1, long ldx2, complex *y, long ldy1, long ldy2, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cfftc3 initializes the trigonometric weight and factor tables or computes the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional complex array. .Ve

```
                    N3-1  N2-1  N1-1
```

Y(k1,k2,k3) = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                    j3=0  j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**

Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed.
- **LDX1 (input)**
  first dimension of X. LDX1 > = N1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. If X and Y are the same array, LDY1 = LDX1 Else LDY1 > = N1 Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least (2*MAX(N,N2,N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1)

  -6 = (LDX2 < N2)

  -7 = (LDY1 < N1) or (LDY1 not equal LDX1 when X and Y are same array)

  -8 = (LDY2 < N2) or (LDY2 not equal LDX2 when X and Y are same array)

  -9 = (LWORK not equal 0) and (LWORK < (2*MAX(N,N2,N3) + 16*N3) * NCPUS)

-10 = memory allocation failed

SEE ALSO

fft

CAUTIONS

This routine uses `Y(N1+1:LDY1,:,:)` as scratch space. Therefore, the original contents of this subarray will be lost upon returning from routine while subarray `Y(1:N1,1:N2,1:N3)` contains the transform results.

NAME

cfftcm - initialize the trigonometric weight and factor tables or compute the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTCM( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)

 SUBROUTINE CFFTCM_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X, Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFTM_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X, Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void cfftcm(int iopt, int n1, int n2, float scale, complex *x, int ldx, complex *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cfftcm_64(long iopt, long n1, long n2, float scale, complex *x, long ldx, complex *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cfftcm initializes the trigonometric weight and factor tables or computes the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional complex array: .Ve

```
        N1-1
```

Y(k,l) = SUM W*X(j,l)

```
        j=0
```

.Ve

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N1)` .Ve

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the input sequences. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying number of input sequences. N2 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  X is a complex array of dimensions (LDX, N2) that contains the sequences to be transformed stored in its columns.
- **LDX (input)**
  Leading dimension of X. LDX > = N1 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY, N2) that contains the transform results of the input sequences. X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = LDX Else LDY > = N1 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*N1 that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N1. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least 2*N1*NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1)

  -5 = (LDY < N1) or (LDY not equal LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < 2*N1*NCPUS)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

cfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of CFFTF followed by a call of CFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE CFFTF( N, X, WSAVE)
COMPLEX X(*)
INTEGER N
REAL WSAVE(*)

SUBROUTINE CFFTF_64( N, X, WSAVE)
COMPLEX X(*)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [N], X, WSAVE)
COMPLEX, DIMENSION(:) :: X
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE FFTF_64( [N], X, WSAVE)
COMPLEX, DIMENSION(:) :: X
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void cfftf(int n, complex *x, float *wsave);

void cfftf_64(long n, complex *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (4 * N + 15) or greater and must have been initialized by CFFTI.

NAME

cffti - initialize the array WSAVE, which is used in both CFFTF and CFFTB.

SYNOPSIS

```
SUBROUTINE CFFTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE CFFTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE CFFTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE CFFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void cffti(int n, float *wsave);

void cffti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (4 * N + 15) or greater. CFFTI needs to be called only once to initialize array WORK before calling CFFTF and/or CFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

cfftopt - compute the length of the closest fast FFT

SYNOPSIS

```
INTEGER FUNCTION CFFTOPT( LEN)
INTEGER LEN

INTEGER*8 FUNCTION CFFTOPT_64( LEN)
INTEGER*8 LEN
```

F95 INTERFACE

```
INTEGER FUNCTION CFFTOPT( LEN)
INTEGER :: LEN

INTEGER(8) FUNCTION CFFTOPT_64( LEN)
INTEGER(8) :: LEN
```

C INTERFACE

#include <sunperf.h>

int cfftopt(int len);

long cfftopt_64(long len);

PURPOSE

cfftopt computes the length of the closest fast FFT. Fast Fourier transform algorithms, including those used in Performance Library, work best with vector lengths that are products of small primes. For example, an FFT of length 32=2**5 will run faster than an FFT of prime length 31 because 32 is a product of small primes and 31 is not. If your application is such that you can taper or zero pad your vector to a larger length then this function may help you select a better length and run your FFT faster.

CFFTOPT will return an integer no smaller than the input argument N that is the closest number that is the product of small primes. CFFTOPT will return 16 for an input of N=16 and return 18=2*3*3 for an input of N=17.

Note that the length computed here is not guaranteed to be optimal, only to be a product of small primes. Also, the value returned may change as the underlying FFTs become capable of handling larger primes. For example, passing in N=51 today will return 52=2*2*13 rather than 51=3*17 because the FFTs in Performance Library do not have fast radix 17 code. In the future, radix 17 code may be added and then N=51 will return 51.

ARGUMENTS

NAME

cffts - initialize the trigonometric weight and factor tables or compute the inverse Fast Fourier Transform of a complex sequence as follows. =head1 SYNOPSIS

```
 SUBROUTINE CFFTS( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX X(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL Y(*), TRIGS(*), WORK(*)

 SUBROUTINE CFFTS_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX X(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL Y(*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, N, [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX, DIMENSION(:) :: X
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: Y, TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, N, [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX, DIMENSION(:) :: X
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: Y, TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void cffts(int iopt, int n, float scale, complex *x, float *y, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cffts_64(long iopt, long n, float scale, complex *x, float *y, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cffts initializes the trigonometric weight and factor tables or computes the inverse Fast Fourier Transform of a complex sequence as follows: .Ve

```
                N-1
```

`Y(k)` = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

k ranges from 0 to N-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

In complex-to-real transform of length N, the (N/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored. Furthermore, due to symmetries the imaginary of the component of `X(0)` and `X(N/2)` (if N is even in the latter) is assumed to be zero and is not referenced.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**

On entry, X is a complex array whose first (N/2+1) elements are the input sequence to be transformed.

- **Y (output)**
  Real array of dimension at least N that contains the transform results. X and Y may be the same array starting at the same memory location. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **TRIGS (input/output)**
  Real array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

cffts2 - initialize the trigonometric weight and factor tables or compute the two-dimensional inverse Fast Fourier Transform of a two-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTS2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX X(LDX,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL Y(LDY,*), TRIGS(*), WORK(*)

 SUBROUTINE CFFTS2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL Y(LDY,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: Y

 SUBROUTINE FFT2_64( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void cffts2(int iopt, int n1, int n2, float scale, complex *x, int ldx, float *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cffts2_64(long iopt, long n1, long n2, float scale, complex *x, long ldx, float *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cffts2 initializes the trigonometric weight and factor tables or computes the two-dimensional inverse Fast Fourier Transform of a two-dimensional complex array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the rows of the input array. One-dimensional FFTs are then computed along the columns of the intermediate results. .Ve

```
                    N1-1  N2-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j1=0  j2=0
.Ve
```

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

In complex-to-real transform of length N1, the (N1/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.

- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX, N2) that contains input data to be transformed.
- **LDX (input)**
  Leading dimension of X. LDX > = (N1/2 + 1) Unchanged on exit.
- **Y (output)**
  Y is a real array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = 2*LDX Else LDY > = 2*LDX and LDY must be even. Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least MAX(N1,2*N2). The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1/2+1)

  -5 = LDY not equal 2*LDX when X and Y are same array

  -6 = (LDY < 2*LDX or LDY odd) when X and Y are same array

  -7 = (LWORK not equal 0) and (LWORK < `MAX(N1,2*N2))`

  -8 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output array Y(1:LDY, 1:N2) is overwritten.

NAME

cffts3 - initialize the trigonometric weight and factor tables or compute the three-dimensional inverse Fast Fourier Transform of a three-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTS3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX1,LDX2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL Y(LDY1,LDY2,*), TRIGS(*), WORK(*)

 SUBROUTINE CFFTS3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX1,LDX2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL Y(LDY1,LDY2,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, N1, [N2], [N3], [SCALE], X, [LDX1], LDX2, Y,
*       [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: X
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:,:) :: Y

 SUBROUTINE FFT3_64( IOPT, N1, [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: X
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void cffts3(int iopt, int n1, int n2, int n3, float scale, complex *x, int ldx1, int ldx2, float *y, int ldy1, int ldy2, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cffts3_64(long iopt, long n1, long n2, long n3, float scale, complex *x, long ldx1, long ldx2, float *y, long ldy1, long ldy2, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cffts3 initializes the trigonometric weight and factor tables or computes the three-dimensional inverse Fast Fourier Transform of a three-dimensional complex array. .Ve

```
                        N3-1  N2-1  N1-1
```

Y(k1,k2,k3) = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                        j3=0  j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed.
- **LDX1 (input)**
  first dimension of X. LDX1 > = N1/2+1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. If X and Y are the same array, LDY1 = 2*LDX1 Else LDY1 > = 2*LDX1 and LDY1 is even Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least (MAX(N,2*N2,2*N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1/2+1)

  -6 = (LDX2 < N2)

  -7 = LDY1 not equal 2*LDX1 when X and Y are same array

  -8 = (LDY1 < 2*LDX1) or (LDY1 is odd) when X and Y are not same array

  -9 = (LDY2 < N2) or (LDY2 not equal LDX2) when X and Y are same array

  -10 = (LWORK not equal 0) and ((LWORK < `MAX(N,2*N2,2*N3)` + 16*N3)*NCPUS)

-11 = memory allocation failed

SEE ALSO

fft

CAUTIONS

This routine uses `Y(N1+1:LDY1,:,:)` as scratch space. Therefore, the original contents of this subarray will be lost upon returning from routine while subarray `Y(1:N1,1:N2,1:N3)` contains the transform results.

NAME

cfftsm - initialize the trigonometric weight and factor tables or compute the one-dimensional inverse Fast Fourier Transform of a set of complex data sequences stored in a two-dimensional array. =head1 SYNOPSIS

```
 SUBROUTINE CFFTSM( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX X(LDX,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL Y(LDY,*), TRIGS(*), WORK(*)

 SUBROUTINE CFFTSM_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX X(LDX,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL Y(LDY,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: Y

 SUBROUTINE FFTM_64( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: X
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void cfftsm(int iopt, int n1, int n2, float scale, complex *x, int ldx, float *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void cfftsm_64(long iopt, long n1, long n2, float scale, complex *x, long ldx, float *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

cfftsm initializes the trigonometric weight and factor tables or computes the one-dimensional inverse Fast Fourier Transform of a set of complex data sequences stored in a two-dimensional array: .Ve

```
                    N1-1
```

`Y(k,l)` = scale * SUM W*X(j,l)

```
                    j=0
```

.Ve

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W = `exp(isign*i*j*k*2*pi/N1)`

In complex-to-real transform of length N1, the (N1/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored. Furthermore, due to symmetries the imaginary of the component of `X(0,0:N2-1)` and `X(N1/2,0:N2-1)` (if N1 is even in the latter) is assumed to be zero and is not referenced.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the input sequences. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying number of input sequences. N2 > = 0. Unchanged on exit.

- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX, N2) that contains the sequences to be transformed stored in its columns in X(0:N1/2, 0:N2-1).
- **LDX (input)**
  Leading dimension of X. LDX > = (N1/2+1) Unchanged on exit.
- **Y (output)**
  Y is a real array of dimensions (LDY, N2) that contains the transform results of the input sequences in Y(0:N1-1,0:N2-1). X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = 2*LDX Else LDY > = N1 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*N1 that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N1. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least N1. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1/2+1)

  -5 = (LDY < N1) or (LDY not equal 2*LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < N1)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

cgbbrd - reduce a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation

---

SYNOPSIS

```
 SUBROUTINE CGBBRD( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q, LDQ,
*      PT, LDPT, C, LDC, WORK, RWORK, INFO)
 CHARACTER * 1 VECT
 COMPLEX AB(LDAB,*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
 INTEGER M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL D(*), E(*), RWORK(*)

 SUBROUTINE CGBBRD_64( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q,
*      LDQ, PT, LDPT, C, LDC, WORK, RWORK, INFO)
 CHARACTER * 1 VECT
 COMPLEX AB(LDAB,*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL D(*), E(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBBRD( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D, E,
*       Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Q, PT, C
 INTEGER :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL, DIMENSION(:) :: D, E, RWORK

 SUBROUTINE GBBRD_64( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D,
*       E, Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Q, PT, C
 INTEGER(8) :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL, DIMENSION(:) :: D, E, RWORK
```

C INTERFACE

#include <sunperf.h>

void cgbbrd(char vect, int m, int n, int ncc, int kl, int ku, complex *ab, int ldab, float *d, float *e, complex *q, int ldq, complex *pt, int ldpt, complex *c, int ldc, int *info);

void cgbbrd_64(char vect, long m, long n, long ncc, long kl, long ku, complex *ab, long ldab, float *d, float *e, complex *q, long ldq, complex *pt, long ldpt, complex *c, long ldc, long *info);

PURPOSE

cgbbrd reduces a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation: Q' * A * P = B.

The routine computes B, and optionally forms Q or P', or computes Q'*C for a given matrix C.

ARGUMENTS

- **VECT (input)**
  Specifies whether or not the matrices Q and P' are to be formed. = 'N': do not form Q or P';

  = 'Q': form Q only;

  = 'P': form P' only;

  = 'B': form both.
- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC >= 0.
- **KL (input)**
  The number of subdiagonals of the matrix A. KL >= 0.
- **KU (input)**
  The number of superdiagonals of the matrix A. KU >= 0.
- **AB (input/output)**
  On entry, the m-by-n band matrix A, stored in rows 1 to KL+KU+1. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl). On exit, A is overwritten by values generated during the reduction.
- **LDAB (input)**
  The leading dimension of the array A. LDAB >= KL+KU+1.
- **D (output)**
  The diagonal elements of the bidiagonal matrix B.
- **E (output)**

The superdiagonal elements of the bidiagonal matrix B.

- **Q (output)**
  If VECT = 'Q' or 'B', the m-by-m unitary matrix Q. If VECT = 'N' or 'P', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,M)` if VECT = 'Q' or 'B'; LDQ >= 1 otherwise.
- **PT (output)**
  If VECT = 'P' or 'B', the n-by-n unitary matrix P'. If VECT = 'N' or 'Q', the array PT is not referenced.
- **LDPT (input)**
  The leading dimension of the array PT. LDPT >= `max(1,N)` if VECT = 'P' or 'B'; LDPT >= 1 otherwise.
- **C (input/output)**
  On entry, an m-by-ncc matrix C. On exit, C is overwritten by Q'*C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)` if NCC > 0; LDC >= 1 if NCC = 0.
- **WORK (workspace)**
  `dimension(MAX(M,N))`
- **RWORK (workspace)**
  `dimension(MAX(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

cgbcon - estimate the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm,

---

SYNOPSIS

```
 SUBROUTINE CGBCON( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NSUB, NSUPER, LDA, INFO
 INTEGER IPIVOT(*)
 REAL ANORM, RCOND
 REAL WORK2(*)

 SUBROUTINE CGBCON_64( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, LDA, INFO
 INTEGER*8 IPIVOT(*)
 REAL ANORM, RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBCON( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT, ANORM,
*       RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, NSUB, NSUPER, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GBCON_64( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT,
*       ANORM, RCOND, [WORK], [WORK2], [INFO])
```

```
CHARACTER(LEN=1) :: NORM
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgbcon(char norm, int n, int nsub, int nsuper, complex *a, int lda, int *ipivot, float anorm, float *rcond, int *info);

void cgbcon_64(char norm, long n, long nsub, long nsuper, complex *a, long lda, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

cgbcon estimates the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA >= 2*NSUB+NSUPER+1.
- **IPIVOT (input)**

The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).

- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  dimension (N)
- **INFO (output)**

```
= 0:  successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE CGBEQU( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC, ROWCND,
*       COLCND, AMAX, INFO)
 COMPLEX A(LDA,*)
 INTEGER M, N, NSUB, NSUPER, LDA, INFO
 REAL ROWCND, COLCND, AMAX
 REAL ROWSC(*), COLSC(*)

 SUBROUTINE CGBEQU_64( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC,
*       ROWCND, COLCND, AMAX, INFO)
 COMPLEX A(LDA,*)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
 REAL ROWCND, COLCND, AMAX
 REAL ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GBEQU( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCND, COLCND, AMAX, [INFO])
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
 REAL :: ROWCND, COLCND, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC

 SUBROUTINE GBEQU_64( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCND, COLCND, AMAX, [INFO])
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
 REAL :: ROWCND, COLCND, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
```

C INTERFACE

#include <sunperf.h>

void cgbequ(int m, int n, int nsub, int nsuper, complex *a, int lda, float *rowsc, float *colsc, float *rowcnd, float *colcnd, float *amax, int *info);

void cgbequ_64(long m, long n, long nsub, long nsuper, complex *a, long lda, float *rowsc, float *colsc, float *rowcnd, float *colcnd, float *amax, long *info);

PURPOSE

cgbequ computes row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input)**
  The band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **ROWSC (output)**
  If INFO = 0, or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCND (output)**
  If INFO = 0 or INFO > M, ROWCND contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCND >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCND (output)**
  If INFO = 0, COLCND contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCND >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, and i is

  < = M: the i-th row of A is exactly zero

  > M: the (i-M)-th column of A is exactly zero

NAME

cgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE CGBMV( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER M, N, NSUB, NSUPER, LDA, INCX, INCY

 SUBROUTINE CGBMV_64( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE GBMV( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, NSUB, NSUPER, LDA, INCX, INCY

 SUBROUTINE GBMV_64( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void cgbmv(char transa, int m, int n, int nsub, int nsuper, complex alpha, complex *a, int lda, complex *x, int incx, complex beta, complex *y, int incy);

void cgbmv_64(char transa, long m, long n, long nsub, long nsuper, complex alpha, complex *a, long lda, complex *x, long incx, complex beta, complex *y, long incy);

PURPOSE

cgbmv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with nsub sub-diagonals and nsuper super-diagonals.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*conjg(A')*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M must be at least zero. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N must be at least zero. Unchanged on exit.
- **NSUB (input)**
  On entry, NSUB specifies the number of sub-diagonals of the matrix A. NSUB must satisfy 0 .le. NSUB. Unchanged on exit.
- **NSUPER (input)**
  On entry, NSUPER specifies the number of super-diagonals of the matrix A. NSUPER must satisfy 0 .le. NSUPER. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading (nsub + nsuper + 1) by n part of the array A must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (nsuper + 1) of the array, the first super-diagonal starting at position 2 in row nsuper, the first sub-diagonal starting at position 1 in row (nsuper + 2), and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left nsuper by nsuper triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
```

```
      K = NSUPER + 1 - J
      DO 10, I = MAX( 1, J - NSUPER ), MIN( M, J + NSUB )
        A( K + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA must be at least (nsub + nsuper + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CGBRFS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF, LDAF,
*       IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGBRFS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*       LDAF, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBRFS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GBRFS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
```

```
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, AF, B, X
INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgbrfs(char transa, int n, int nsub, int nsuper, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cgbrfs_64(char transa, long n, long nsub, long nsuper, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cgbrfs improves the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The original band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(n,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **AF (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during

the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB*NSUPER+1.
- **IPIVOT (input)**
  The pivot indices from CGBTRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CGBTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgbsv - compute the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE CGBSV( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*      INFO)
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE CGBSV_64( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*      INFO)
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSV( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GBSV_64( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgbsv(int n, int nsub, int nsuper, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void cgbsv_64(long n, long nsub, long nsuper, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cgbsv computes the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = L * U, where L is a product of permutation and unit lower triangular matrices with KL subdiagonals, and U is upper triangular with KL+KU superdiagonals. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUB+NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+NSUB) On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +       *    *    *   u14  u25  u36
 *    *    +    +    +    +       *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *      m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *      m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

cgbsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE CGBSVX( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*       LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR,
*       BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGBSVX_64( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA,
*       AF, LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSVX( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

```
 SUBROUTINE GBSVX_64( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A,
*       [LDA], AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgbsvx(char fact, char transa, int n, int nsub, int nsuper, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, char equed, float *rowsc, float *colsc, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cgbsvx_64(char fact, char transa, long n, long nsub, long nsuper, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, char equed, float *rowsc, float *colsc, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cgbsvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed by this subroutine:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = L * U,

where L is a product of permutation and unit lower triangular
matrices with KL subdiagonals, and U is upper triangular with
KL+KU superdiagonals.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine

precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```
- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+kl)

  If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains details of the LU factorization of the band matrix

A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

If FACT = 'N', then AF is an output argument and on exit returns details of the LU factorization of A.

If FACT = 'E', then AF is an output argument and on exit returns details of the LU factorization of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB+NSUPER+1.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = L*U as computed by CGBTRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the equilibrated matrix A.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(2*N)
- **WORK2 (workspace)**
  dimension(N) On exit, WORK2(1) contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If WORK2(1) is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then WORK2(1) contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

cgbtf2 - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE CGBTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO)
COMPLEX AB(LDAB,*)
INTEGER M, N, KL, KU, LDAB, INFO
INTEGER IPIV(*)

SUBROUTINE CGBTF2_64( M, N, KL, KU, AB, LDAB, IPIV, INFO)
COMPLEX AB(LDAB,*)
INTEGER*8 M, N, KL, KU, LDAB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE GBTF2( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
COMPLEX, DIMENSION(:,:) :: AB
INTEGER :: M, N, KL, KU, LDAB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE GBTF2_64( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
COMPLEX, DIMENSION(:,:) :: AB
INTEGER(8) :: M, N, KL, KU, LDAB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void cgbtf2(int m, int n, int kl, int ku, complex *ab, int ldab, int *ipiv, int *info);

void cgbtf2_64(long m, long n, long kl, long ku, complex *ab, long ldab, long *ipiv, long *info);

PURPOSE

cgbtf2 computes an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **KL (input)**
  The number of subdiagonals within the band of A. KL >= 0.
- **KU (input)**
  The number of superdiagonals within the band of A. KU >= 0.
- **AB (input/output)**
  On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1. See below for further details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= 2*KL+KU+1.
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, KL = 2, KU = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U, because of fill-in resulting from the row

interchanges.

NAME

cgbtrf - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE CGBTRF( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
COMPLEX A(LDA,*)
INTEGER M, N, NSUB, NSUPER, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGBTRF_64( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
COMPLEX A(LDA,*)
INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GBTRF( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GBTRF_64( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgbtrf(int m, int n, int nsub, int nsuper, complex *a, int lda, int *ipivot, int *info);

void cgbtrf_64(long m, long n, long nsub, long nsuper, complex *a, long lda, long *ipivot, long *info);

PURPOSE

cgbtrf computes an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

cgbtrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF

SYNOPSIS

```
 SUBROUTINE CGBTRS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE CGBTRS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GBTRS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GBTRS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgbtrs(char transa, int n, int nsub, int nsuper, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void cgbtrs_64(char transa, long n, long nsub, long nsuper, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cgbtrs solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgebak - form the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL

SYNOPSIS

```
 SUBROUTINE CGEBAK( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 COMPLEX V(LDV,*)
 INTEGER N, ILO, IHI, M, LDV, INFO
 REAL SCALE(*)

 SUBROUTINE CGEBAK_64( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV,
*      INFO)
 CHARACTER * 1 JOB, SIDE
 COMPLEX V(LDV,*)
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 REAL SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBAK( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX, DIMENSION(:,:) :: V
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: SCALE

 SUBROUTINE GEBAK_64( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX, DIMENSION(:,:) :: V
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void cgebak(char job, char side, int n, int ilo, int ihi, float *scale, int m, complex *v, int ldv, int *info);

void cgebak_64(char job, char side, long n, long ilo, long ihi, float *scale, long m, complex *v, long ldv, long *info);

PURPOSE

cgebak forms the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required: = 'N', do nothing, return immediately; = 'P', do backward transformation for permutation only; = 'S', do backward transformation for scaling only; = 'B', do backward transformations for both permutation and scaling. JOB must be the same as the argument JOB supplied to CGEBAL.
- **SIDE (input)**

  ```
  = 'R':  V contains right eigenvectors;

  = 'L':  V contains left eigenvectors.
  ```
- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integer ILO determined by CGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  The integer IHI determined by CGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **SCALE (input)**
  Details of the permutation and scaling factors, as returned by CGEBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**
  On entry, the matrix of right or left eigenvectors to be transformed, as returned by CHSEIN or CTREVC. On exit, V is overwritten by the transformed eigenvectors.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

cgebal - balance a general complex matrix A

SYNOPSIS

```
SUBROUTINE CGEBAL( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
COMPLEX A(LDA,*)
INTEGER N, LDA, ILO, IHI, INFO
REAL SCALE(*)

SUBROUTINE CGEBAL_64( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, ILO, IHI, INFO
REAL SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE GEBAL( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, ILO, IHI, INFO
REAL, DIMENSION(:) :: SCALE

SUBROUTINE GEBAL_64( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, ILO, IHI, INFO
REAL, DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void cgebal(char job, int n, complex *a, int lda, int *ilo, int *ihi, float *scale, int *info);

void cgebal_64(char job, long n, complex *a, long lda, long *ilo, long *ihi, float *scale, long *info);

PURPOSE

cgebal balances a general complex matrix A. This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to ILO-1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, SCALE(I)  = 1.0
for i  = 1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **SCALE (output)**
  Details of the permutations and scaling factors applied to A. If `P(j)` is the index of the row and column interchanged with row and column j and `D(j)` is the scaling factor applied to row and column j, then `SCALE(j)` = `P(j)` for j = 1,...,ILO-1 = `D(j)` for j = ILO,...,IHI = `P(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

The permutations consist of row and column interchanges which put the matrix in the form

```
          ( T1   X   Y  )
P A P  = (  0   B   Z  )
          (  0   0   T2 )
```

where T1 and T2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ILO and IHI mark the starting and ending columns of the submatrix B. Balancing consists of applying a diagonal similarity transformation `inv(D)` * B * D to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

```
( T1     X*D          Y    )
(  0  inv(D)*B*D  inv(D)*Z ).
(  0      0           T2   )
```

Information about the permutations P and the diagonal matrix D is returned in the vector SCALE.

This subroutine is based on the EISPACK routine CBAL.

Modified by Tzu-Yi Chen, Computer Science Division, University of California at Berkeley, USA

NAME

cgebrd - reduce a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation

---

SYNOPSIS

```
 SUBROUTINE CGEBRD( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 COMPLEX A(LDA,*), TAUQ(*), TAUP(*), WORK(*)
 INTEGER M, N, LDA, LWORK, INFO
 REAL D(*), E(*)

 SUBROUTINE CGEBRD_64( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 COMPLEX A(LDA,*), TAUQ(*), TAUP(*), WORK(*)
 INTEGER*8 M, N, LDA, LWORK, INFO
 REAL D(*), E(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBRD( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUQ, TAUP, WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, LWORK, INFO
 REAL, DIMENSION(:) :: D, E

 SUBROUTINE GEBRD_64( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUQ, TAUP, WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 REAL, DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void cgebrd(int m, int n, complex *a, int lda, float *d, float *e, complex *tauq, complex *taup, int *info);

void cgebrd_64(long m, long n, complex *a, long lda, float *d, float *e, complex *tauq, complex *taup, long *info);

PURPOSE

cgebrd reduces a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation: Q**H * A * P = B.

If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.

ARGUMENTS

- **M (input)**
  The number of rows in the matrix A. M > = 0.
- **N (input)**
  The number of columns in the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N general matrix to be reduced. On exit, if m > = n, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array TAUQ, represent the unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array TAUP, represent the unitary matrix P as a product of elementary reflectors; if m < n, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array TAUQ, represent the unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the unitary matrix P as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **D (output)**
  The diagonal elements of the bidiagonal matrix B: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the bidiagonal matrix B: if m > = n, `E(i)` = `A(i,i+1)` for i = 1,2,...,n-1; if m < n, `E(i)` = `A(i+1,i)` for i = 1,2,...,m-1.
- **TAUQ (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q. See Further Details.
- **TAUP (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix P. See Further Details.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,M,N). For optimum performance LWORK > = (M+N)*NB, where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrices Q and P are represented as products of elementary reflectors:

If m >= n,

```
Q  = H(1) H(2) . . . H(n)  and  P  = G(1) G(2) . . . G(n-1)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are complex scalars, and v and u are complex vectors; `v(1:i-1)` = 0, `v(i)` = 1, and `v(i+1:m)` is stored on exit in A(i+1:m,i); `u(1:i)` = 0, `u(i+1)` = 1, and `u(i+2:n)` is stored on exit in A(i,i+2:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

If m < n,

```
Q  = H(1) H(2) . . . H(m-1)  and  P  = G(1) G(2) . . . G(m)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are complex scalars, and v and u are complex vectors; `v(1:i)` = 0, `v(i+1)` = 1, and `v(i+2:m)` is stored on exit in A(i+2:m,i); `u(1:i-1)` = 0, `u(i)` = 1, and `u(i+1:n)` is stored on exit in A(i,i+1:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

The contents of A on exit are illustrated by the following examples:

m = 6 and n = 5 (m > n): m = 5 and n = 6 (m < n):

```
(  d   e   u1  u1  u1 )          (  d   u1  u1  u1  u1  u1 )
(  v1  d   e   u2  u2 )          (  e   d   u2  u2  u2  u2 )
(  v1  v2  d   e   u3 )          (  v1  e   d   u3  u3  u3 )
(  v1  v2  v3  d   e  )          (  v1  v2  e   d   u4  u4 )
(  v1  v2  v3  v4  d  )          (  v1  v2  v3  e   d   u5 )
(  v1  v2  v3  v4  v5 )
```

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining H(i), and ui an element of the vector defining G(i).

NAME

cgecon - estimate the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF

SYNOPSIS

```
 SUBROUTINE CGECON( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)

 SUBROUTINE CGECON_64( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GECON( NORM, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GECON_64( NORM, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgecon(char norm, int n, complex *a, int lda, float anorm, float *rcond, int *info);

void cgecon_64(char norm, long n, complex *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

cgecon estimates the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE CGEEQU( M, N, A, LDA, ROWSC, COLSC, ROWCND, COLCND, AMAX,
*      INFO)
 COMPLEX A(LDA,*)
 INTEGER M, N, LDA, INFO
 REAL ROWCND, COLCND, AMAX
 REAL ROWSC(*), COLSC(*)

 SUBROUTINE CGEEQU_64( M, N, A, LDA, ROWSC, COLSC, ROWCND, COLCND,
*      AMAX, INFO)
 COMPLEX A(LDA,*)
 INTEGER*8 M, N, LDA, INFO
 REAL ROWCND, COLCND, AMAX
 REAL ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEQU( [M], [N], A, [LDA], ROWSC, COLSC, ROWCND, COLCND,
*       AMAX, [INFO])
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, INFO
 REAL :: ROWCND, COLCND, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC

 SUBROUTINE GEEQU_64( [M], [N], A, [LDA], ROWSC, COLSC, ROWCND,
*       COLCND, AMAX, [INFO])
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, INFO
 REAL :: ROWCND, COLCND, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
```

C INTERFACE

#include <sunperf.h>

void cgeequ(int m, int n, complex *a, int lda, float *rowsc, float *colsc, float *rowcnd, float *colcnd, float *amax, int *info);

void cgeequ_64(long m, long n, complex *a, long lda, float *rowsc, float *colsc, float *rowcnd, float *colcnd, float *amax, long *info);

PURPOSE

cgeequ computes row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input)**
  The M-by-N matrix whose equilibration factors are to be computed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **ROWSC (output)**
  If INFO = 0 or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCND (output)**
  If INFO = 0 or INFO > M, ROWCND contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCND >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCND (output)**
  If INFO = 0, COLCND contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCND >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: if INFO = i, and i is

< = M: the i-th row of A is exactly zero

> M: the (i-M)-th column of A is exactly zero

NAME

cgees - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE CGEES( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, W, Z, LDZ,
*      WORK, LDWORK, WORK2, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL WORK3(*)
 REAL WORK2(*)

 SUBROUTINE CGEES_64( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, W, Z,
*      LDZ, WORK, LDWORK, WORK2, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 WORK3(*)
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEES( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, W, Z,
*       [LDZ], [WORK], [LDWORK], [WORK2], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: WORK3
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GEES_64( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, W, Z,
*       [LDZ], [WORK], [LDWORK], [WORK2], [WORK3], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, SORTEV
COMPLEX, DIMENSION(:) :: W, WORK
COMPLEX, DIMENSION(:,:) :: A, Z
INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
LOGICAL(8) :: SELECT
LOGICAL(8), DIMENSION(:) :: WORK3
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgees(char jobz, char sortev, logical(*select)(complex), int n, complex *a, int lda, int *nout, complex *w, complex *z, int ldz, int *info);

void cgees_64(char jobz, char sortev, logical(*select)(complex), long n, complex *a, long lda, long *nout, complex *w, complex *z, long ldz, long *info);

PURPOSE

cgees computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**H).

Optionally, it also orders the eigenvalues on the diagonal of the Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A complex matrix is in Schur form if it is upper triangular.

ARGUMENTS

- **JOBZ (input)**

  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered:

  = 'S': Eigenvalues are ordered (see SELECT).

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to order to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. The eigenvalue `W(j)` is selected if `SELECT(W(j))` is true.

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**

On entry, the N-by-N matrix A. On exit, A has been overwritten by its Schur form T.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues for which SELECT is true.
- **W (output)**
  W contains the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T.
- **Z (output)**
  If JOBZ = 'V', Z contains the unitary matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1; if JOBZ = 'V', LDZ >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **WORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N:  the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of W contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the matrix which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT = .TRUE.. This could also be caused by underflow due to scaling.

NAME

cgeesx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE CGEESX( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT, W,
*      Z, LDZ, RCONE, RCONV, WORK, LDWORK, WORK2, BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL BWORK3(*)
 REAL RCONE, RCONV
 REAL WORK2(*)

 SUBROUTINE CGEESX_64( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT,
*      W, Z, LDZ, RCONE, RCONV, WORK, LDWORK, WORK2, BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 BWORK3(*)
 REAL RCONE, RCONV
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEESX( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA], NOUT,
*       W, Z, [LDZ], RCONE, RCONV, [WORK], [LDWORK], [WORK2], [BWORK3],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: BWORK3
 REAL :: RCONE, RCONV
```

```
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GEESX_64( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA],
*       NOUT, W, Z, [LDZ], RCONE, RCONV, [WORK], [LDWORK], [WORK2],
*       [BWORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL(8) :: SELECT
 LOGICAL(8), DIMENSION(:) :: BWORK3
 REAL :: RCONE, RCONV
 REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgeesx(char jobz, char sortev, logical(*select)(complex), char sense, int n, complex *a, int lda, int *nout, complex *w, complex *z, int ldz, float *rcone, float *rconv, int *info);

void cgeesx_64(char jobz, char sortev, logical(*select)(complex), char sense, long n, complex *a, long lda, long *nout, complex *w, complex *z, long ldz, float *rcone, float *rconv, long *info);

PURPOSE

cgeesx computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**H).

Optionally, it also orders the eigenvalues on the diagonal of the Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (RCONDV). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see Section 4.10 of the LAPACK Users' Guide (where these quantities are called s and sep respectively).

A complex matrix is in Schur form if it is upper triangular.

ARGUMENTS

- **JOBZ (input)**

  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S': Eigenvalues are ordered (see SELECT).
```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to order to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `W(j)` is selected if `SELECT(W(j))` is true.
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
= 'E': Computed for average of selected eigenvalues only;

= 'V': Computed for selected right invariant subspace only;

= 'B': Computed for both.
If SENSE  = 'E', 'V' or 'B', SORTEV must equal 'S'.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A is overwritten by its Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues for which SELECT is true.
- **W (output)**
  W contains the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T.
- **Z (output)**
  If JOBZ = 'V', Z contains the unitary matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.
- **RCONE (output)**
  If SENSE = 'E' or 'B', RCONE contains the reciprocal condition number for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONV (output)**
  If SENSE = 'V' or 'B', RCONV contains the reciprocal condition number for the selected right invariant subspace. Not referenced if SENSE = 'N' or 'E'.
- **WORK (workspace)**
  `dimension(LDWORK)` On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). Also, if SENSE = 'E' or 'V' or 'B', LDWORK >= 2*NOUT*(N-NOUT), where NOUT is the number of selected eigenvalues computed by this routine. Note that 2*NOUT*(N-NOUT) <= N*N/2. For good performance, LDWORK must generally be larger.
- **WORK2 (workspace)**
  `dimension(N)`
- **BWORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.
```

> 0: if INFO = i, and i is

< = N: the QR algorithm failed to compute all the

eigenvalues; elements 1:ILO-1 and i+1:N of W contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the transformation which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

cgeev - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE CGEEV( JOBVL, JOBVR, N, A, LDA, W, VL, LDVL, VR, LDVR,
*      WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL WORK2(*)

 SUBROUTINE CGEEV_64( JOBVL, JOBVR, N, A, LDA, W, VL, LDVL, VR, LDVR,
*      WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEV( JOBVL, JOBVR, [N], A, [LDA], W, VL, [LDVL], VR,
*       [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, VL, VR
 INTEGER :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GEEV_64( JOBVL, JOBVR, [N], A, [LDA], W, VL, [LDVL], VR,
*       [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, VL, VR
 INTEGER(8) :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgeev(char jobvl, char jobvr, int n, complex *a, int lda, complex *w, complex *vl, int ldvl, complex *vr, int ldvr, int *info);

void cgeev_64(char jobvl, char jobvr, long n, complex *a, long lda, complex *w, complex *vl, long ldvl, complex *vr, long ldvr, long *info);

PURPOSE

cgeev computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N': left eigenvectors of A are not computed;

  = 'V': left eigenvectors of are computed.
  ```

- **JOBVR (input)**

  ```
  = 'N': right eigenvectors of A are not computed;

  = 'V': right eigenvectors of A are computed.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **W (output)**
  W contains the computed eigenvalues.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. `u(j)` = VL(:,j), the j-th column of VL.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. `v(j)` = VR(:,j), the j-th column of VR.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors have been computed;
elements and i+1:N of W contain eigenvalues which have
converged.
```

NAME

cgeevx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE CGEEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, W, VL,
*      LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL ABNRM
 REAL SCALE(*), RCONE(*), RCONV(*), WORK2(*)

 SUBROUTINE CGEEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, W, VL,
*      LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL ABNRM
 REAL SCALE(*), RCONE(*), RCONV(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], W, VL,
*       [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV, [WORK],
*       [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, VL, VR
 INTEGER :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL :: ABNRM
 REAL, DIMENSION(:) :: SCALE, RCONE, RCONV, WORK2

 SUBROUTINE GEEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], W,
*       VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
```

```
*       [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: A, VL, VR
 INTEGER(8) :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL :: ABNRM
 REAL, DIMENSION(:) :: SCALE, RCONE, RCONV, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgeevx(char balanc, char jobvl, char jobvr, char sense, int n, complex *a, int lda, complex *w, complex *vl, int ldvl, complex *vr, int ldvr, int *ilo, int *ihi, float *scale, float *abnrm, float *rcone, float *rconv, int *info);

void cgeevx_64(char balanc, char jobvl, char jobvr, char sense, long n, complex *a, long lda, complex *w, complex *vl, long ldvl, complex *vr, long ldvr, long *ilo, long *ihi, float *scale, float *abnrm, float *rcone, float *rconv, long *info);

PURPOSE

cgeevx computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, SCALE, and ABNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right

eigenvectors (RCONDV).

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D**(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide.

ARGUMENTS

- **BALANC (input)**
  Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. = 'N': Do not diagonally scale or permute;

```
 = 'P': Perform permutations to make the matrix more nearly
upper triangular. Do not diagonally scale;
 = 'S': Diagonally scale the matrix, ie. replace A by
D*A*D**(-1), where D is a diagonal matrix chosen
to make the rows and columns of A more equal in
norm. Do not permute;
 = 'B': Both diagonally scale and permute A.
```

  Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

- **JOBVL (input)**

```
 = 'N': left eigenvectors of A are not computed;

 = 'V': left eigenvectors of A are computed.
If SENSE  = 'E' or 'B', JOBVL must  = 'V'.
```

- **JOBVR (input)**

```
 = 'N': right eigenvectors of A are not computed;

 = 'V': right eigenvectors of A are computed.
If SENSE  = 'E' or 'B', JOBVR must  = 'V'.
```

- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
 = 'E': Computed for eigenvalues only;

 = 'V': Computed for right eigenvectors only;

 = 'B': Computed for eigenvalues and right eigenvectors.
```

  If SENSE = 'E' or 'B', both left and right eigenvectors must also be computed (JOBVL = 'V' and JOBVR = 'V').

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten. If JOBVL = 'V' or JOBVR = 'V', A contains the Schur form of the balanced version of the matrix A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  W contains the computed eigenvalues.

- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. `u(j)` = VL(:,j), the j-th column of VL.

- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.

- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. `v(j)` = VR(:,j), the j-th column of VR.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced `A(i,j)` = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **IHI (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced `A(i,j)` = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **SCALE (output)**
  Details of the permutations and scaling factors applied when balancing A. If `P(j)` is the index of the row and column interchanged with row and column j, and `D(j)` is the scaling factor applied to row and column j, then `SCALE(J)` = P(J), for J = 1,...,ILO-1 = D(J), for J = ILO,...,IHI = `P(J)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
- **RCONE (output)**
  `RCONE(j)` is the reciprocal condition number of the j-th eigenvalue.
- **RCONV (output)**
  `RCONV(j)` is the reciprocal condition number of the j-th right eigenvector.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. If SENSE = 'N' or 'E', LDWORK >= max(1,2*N), and if SENSE = 'V' or 'B', LDWORK >= N*N+2*N. For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors or condition numbers
have been computed; elements 1:ILO-1 and i+1:N of W
contain eigenvalues which have converged.
```

NAME

cgegs - routine is deprecated and has been replaced by routine CGGES

SYNOPSIS

```
 SUBROUTINE CGEGS( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL WORK2(*)

 SUBROUTINE CGEGS_64( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGS( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GEGS_64( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER(8) :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgegs(char jobvsl, char jobvsr, int n, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *vsl, int ldvsl, complex *vsr, int ldvsr, int *info);

void cgegs_64(char jobvsl, char jobvsr, long n, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *vsl, long ldvsl, complex *vsr, long ldvsr, long *info);

PURPOSE

cgegs routine is deprecated and has been replaced by routine CGGES.

CGEGS computes for a pair of N-by-N complex nonsymmetric matrices A, B: the generalized eigenvalues (alpha, beta), the complex Schur form (A, B), and optionally left and/or right Schur vectors (VSL and VSR).

(If only the generalized eigenvalues are needed, use the driver CGEGV instead.)

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations", by G. Golub & C. van Loan (Johns Hopkins U. Press)

The (generalized) Schur form of a pair of matrices is the result of multiplying both matrices on the left by one unitary matrix and both on the right by another unitary matrix, these two unitary matrices being chosen so as to bring the pair of matrices into upper triangular form with the diagonal elements of B being non-negative real numbers (this is also called complex Schur form.)

The left and right Schur vectors are the columns of VSL and VSR, respectively, where VSL and VSR are the unitary matrices

which reduce A and B to Schur form:

Schur form of (A,B) = ((VSL)**H A (VSR), (VSL)**H B (VSR))

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **N (input)**

The order of the matrices A, B, VSL, and VSR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of A.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of B.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHA(j), j =1,...,N and BETA(j), j =1,...,N are the diagonals of the complex Schur form (A,B) output by CGEGS. The `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See the description of ALPHA.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > = 1, and if JOBVSL = 'V', LDVSL > = N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR > = 1, and if JOBVSR = 'V', LDVSR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,2*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for CGEQRF, CUNMQR, and CUNGQR.) Then compute: NB as the MAX of the blocksizes for CGEQRF, CUNMQR, and CUNGQR; the optimal LDWORK is N*(NB+1).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
```

```
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from CGGBAL

 =N+2: error return from CGEQRF

 =N+3: error return from CUNMQR

 =N+4: error return from CUNGQR

 =N+5: error return from CGGHRD

 =N+6: error return from CHGEQZ (other than failed
iteration)
 =N+7: error return from CGGBAK (computing VSL)

 =N+8: error return from CGGBAK (computing VSR)

 =N+9: error return from CLASCL (various places)
```

NAME

cgegv - routine is deprecated and has been replaced by routine CGGEV

SYNOPSIS

```
 SUBROUTINE CGEGV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA, VL,
*      LDVL, VR, LDVR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL WORK2(*)

 SUBROUTINE CGEGV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VL, LDVL, VR, LDVR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA, BETA,
*       VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GEGV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgegv(char jobvl, char jobvr, int n, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *vl, int ldvl, complex *vr, int ldvr, int *info);

void cgegv_64(char jobvl, char jobvr, long n, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *vl, long ldvl, complex *vr, long ldvr, long *info);

PURPOSE

cgegv routine is deprecated and has been replaced by routine CGGEV.

CGEGV computes for a pair of N-by-N complex nonsymmetric matrices A and B, the generalized eigenvalues (alpha, beta), and optionally, the left and/or right generalized eigenvectors (VL and VR).

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations", by G. Golub & C. van Loan (Johns Hopkins U. Press)

A right generalized eigenvector corresponding to a generalized eigenvalue w for a pair of matrices (A,B) is a vector r such that (A - w B) r = 0 . A left generalized eigenvector is a vector l such that l**H * (A - w B) = 0, where l**H is the

conjugate-transpose of l.

Note: this routine performs ``full balancing" on A and B. See ``Further Details", below.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, BETA, and VR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of A on exit, see ``Further Details", below.)

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of B on exit, see ``Further Details'', below.)
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix BETA. LDVL > = 1, and if JOBVL = 'V', LDVL > = N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR > = 1, and if JOBVR = 'V', LDVR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,2*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for CGEQRF, CUNMQR, and CUNGQR.) Then compute: NB as the MAX of the blocksizes for CGEQRF, CUNMQR, and CUNGQR; The optimal LDWORK is MAX(2*N, N*(NB+1)).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and BETA(j) should be
correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:
```

```
 =N+1: error return from CGGBAL

 =N+2: error return from CGEQRF

 =N+3: error return from CUNMQR

 =N+4: error return from CUNGQR

 =N+5: error return from CGGHRD

 =N+6: error return from CHGEQZ (other than failed
iteration)
 =N+7: error return from CTGEVC

 =N+8: error return from CGGBAK (computing BETA)

 =N+9: error return from CGGBAK (computing VR)

 =N+10: error return from CLASCL (various calls)
```

FURTHER DETAILS

Balancing

This driver calls CGGBAL to both permute and scale rows and columns of A and B. The permutations PL and PR are chosen so that PL*A*PR and PL*B*R will be upper triangular except for the diagonal blocks `A(i:j,i:j)` and B(i:j,i:j), with i and j as close together as possible. The diagonal scaling matrices DL and DR are chosen so that the pair DL*PL*A*PR*DR, DL*PL*B*PR*DR have elements close to one (except for the elements that start out zero.)

After the eigenvalues and eigenvectors of the balanced matrices have been computed, CGGBAK transforms the eigenvectors back to what they would have been (in perfect arithmetic) if they had not been balanced.

Contents of A and B on Exit

-------- -- - --- - -- ----

If any eigenvectors are computed (either JOBVL ='V' or JOBVR ='V' or both), then on exit the arrays A and B will contain the complex Schur form[*] of the ``balanced'' versions of A and B. If no eigenvectors are computed, then only the diagonal blocks will be correct.

[*] In other words, upper triangular form.

NAME

cgehrd - reduce a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation

---

SYNOPSIS

```
  SUBROUTINE CGEHRD( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN, INFO)
  COMPLEX A(LDA,*), TAU(*), WORKIN(*)
  INTEGER N, ILO, IHI, LDA, LWORKIN, INFO

  SUBROUTINE CGEHRD_64( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN,
 *      INFO)
  COMPLEX A(LDA,*), TAU(*), WORKIN(*)
  INTEGER*8 N, ILO, IHI, LDA, LWORKIN, INFO
```

F95 INTERFACE

```
  SUBROUTINE GEHRD( [N], ILO, IHI, A, [LDA], TAU, [WORKIN], [LWORKIN],
 *       [INFO])
  COMPLEX, DIMENSION(:) :: TAU, WORKIN
  COMPLEX, DIMENSION(:,:) :: A
  INTEGER :: N, ILO, IHI, LDA, LWORKIN, INFO

  SUBROUTINE GEHRD_64( [N], ILO, IHI, A, [LDA], TAU, [WORKIN],
 *       [LWORKIN], [INFO])
  COMPLEX, DIMENSION(:) :: TAU, WORKIN
  COMPLEX, DIMENSION(:,:) :: A
  INTEGER(8) :: N, ILO, IHI, LDA, LWORKIN, INFO
```

C INTERFACE

#include <sunperf.h>

void cgehrd(int n, int ilo, int ihi, complex *a, int lda, complex *tau, int *info);

void cgehrd_64(long n, long ilo, long ihi, complex *a, long lda, complex *tau, long *info);

PURPOSE

cgehrd reduces a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation: Q' * A * Q = H .

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGEBAL; otherwise they should be set to 1 and N respectively. See Further Details.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details). Elements 1:ILO-1 and IHI:N-1 of TAU are set to zero.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The length of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of (ihi-ilo) elementary reflectors

```
Q  = H(ilo) H(ilo+1) . . . H(ihi-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0, `v(i+1)` = 1 and `v(ihi+1:n)` = 0; `v(i+2:ihi)` is stored on exit in A(i+2:ihi,i), and tau in TAU(i).

The contents of A are illustrated by the following example, with n = 7, ilo = 2 and ihi = 6:

on entry, on exit,

(a a a a a a a) (a a h h h h a) (a a a a a a) (a h h h h a) (a a a a a a) (h h h h h h) (a a a a a a) (v2 h h h h h) (a a a a a a) (v2 v3 h h h h) (a a a a a a) (v2 v3 v4 h h h) (a) (a)

where a denotes an element of the original matrix A, h denotes a modified element of the upper Hessenberg matrix H, and vi denotes an element of the vector defining H(i).

NAME

cgelqf - compute an LQ factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE CGELQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE CGELQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GELQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GELQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgelqf(int m, int n, complex *a, int lda, complex *tau, int *info);

void cgelqf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

cgelqf computes an LQ factorization of a complex M-by-N matrix A: A = L * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and below the diagonal of the array contain the m-by-min(m,n) lower trapezoidal matrix L (L is lower triangular if m <= n); the elements above the diagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k)' . . . H(2)' H(1)', where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `conjg(v(i+1:n))` is stored

on exit in A(i,i+1:n), and tau in TAU(i).

NAME

cgels - solve overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A

SYNOPSIS

```
 SUBROUTINE CGELS( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK, LDWORK,
*      INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, LDWORK, INFO

 SUBROUTINE CGELS_64( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GELS( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, LDWORK, INFO

 SUBROUTINE GELS_64( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgels(char transa, int m, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void cgels_64(char transa, long m, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

cgels solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||.

2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B.

3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B.

4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

ARGUMENTS

- **TRANSA (input)**

  ```
  = 'N': the linear system involves A;

  = 'C': the linear system involves A**H.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. if M > = N, A is overwritten by details of its QR factorization as returned by CGEQRF; if M < N, A is overwritten by details of its LQ factorization as returned by CGELQF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the matrix B of right hand side vectors, stored columnwise; B is M-by-NRHS if TRANSA = 'N', or

N-by-NRHS if TRANSA = 'C'. On exit, B is overwritten by the solution vectors, stored columnwise: if TRANSA = 'N' and m >= n, rows 1 to n of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements N+1 to M in that column; if TRANSA = 'N' and m < n, rows 1 to N of B contain the minimum norm solution vectors; if TRANSA = 'C' and m >= n, rows 1 to M of B contain the minimum norm solution vectors; if TRANSA = 'C' and m < n, rows 1 to M of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements M+1 to N in that column.

- **LDB (input)**
  The leading dimension of the array B. LDB >= MAX(1,M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1, MN + max(MN, NRHS)). For optimal performance, LDWORK >= max(1, MN + max(MN, NRHS)*NB). where MN = `min(M,N)` and NB is the optimum block size.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgelsd - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE CGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK,
*      LWORK, RWORK, IWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER IWORK(*)
 REAL RCOND
 REAL S(*), RWORK(*)

 SUBROUTINE CGELSD_64( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK,
*      WORK, LWORK, RWORK, IWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL RCOND
 REAL S(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSD( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: RCOND
 REAL, DIMENSION(:) :: S, RWORK

 SUBROUTINE GELSD_64( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL :: RCOND
 REAL, DIMENSION(:) :: S, RWORK
```

C INTERFACE

#include <sunperf.h>

void cgelsd(int m, int n, int nrhs, complex *a, int lda, complex *b, int ldb, float *s, float rcond, int *rank, int *info);

void cgelsd_64(long m, long n, long nrhs, complex *a, long lda, complex *b, long ldb, float *s, float rcond, long *rank, long *info);

PURPOSE

cgelsd computes the minimum-norm solution to a real linear least squares problem: minimize 2-norm(| b - A*x |)

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The problem is solved in three steps:

(1) Reduce the coefficient matrix A to bidiagonal form with Householder tranformations, reducing the original problem into a ``bidiagonal least squares problem'' (BLS)

(2) Solve the BLS using a divide and conquer approach.

(3) Apply back all the Householder tranformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and RANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,M,N).
- **S (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = S(1)/S(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `S(i)` < = RCOND*S(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **RANK (output)**

The effective rank of A, i.e., the number of singular values which are greater than RCOND*S(1).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. The exact minimum amount of workspace needed depends on M, N and NRHS. If M >= N, LWORK >= 2*N + N*NRHS. If M < N, LWORK >= 2*M + M*NRHS. For good performance, LWORK should generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  If M >= N, LRWORK >= 8*N + 2*N*SMLSIZ + 8*N*NLVL + N*NRHS. If M < N, LRWORK >= 8*M + 2*M*SMLSIZ + 8*M*NLVL + M*NRHS. SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and NLVL = INT(LOG_2(MIN(M,N)/(SMLSIZ+1))) + 1
- **IWORK (workspace)**
  LIWORK >= 3 * MINMN * NLVL + 11 * MINMN, where MINMN = MIN(M,N).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Ren-Cang Li, Computer Science Division, University of California at Berkeley, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

cgelss - compute the minimum norm solution to a complex linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE CGELSS( M, N, NRHS, A, LDA, B, LDB, SING, RCOND, IRANK,
*      WORK, LDWORK, WORK2, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL RCOND
 REAL SING(*), WORK2(*)

 SUBROUTINE CGELSS_64( M, N, NRHS, A, LDA, B, LDB, SING, RCOND,
*      IRANK, WORK, LDWORK, WORK2, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL RCOND
 REAL SING(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSS( [M], [N], [NRHS], A, [LDA], B, [LDB], SING, RCOND,
*       IRANK, [WORK], [LDWORK], [WORK2], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SING, WORK2

 SUBROUTINE GELSS_64( [M], [N], [NRHS], A, [LDA], B, [LDB], SING,
*       RCOND, IRANK, [WORK], [LDWORK], [WORK2], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SING, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgelss(int m, int n, int nrhs, complex *a, int lda, complex *b, int ldb, float *sing, float rcond, int *irank, int *info);

void cgelss_64(long m, long n, long nrhs, complex *a, long lda, complex *b, long ldb, float *sing, float rcond, long *irank, long *info);

PURPOSE

cgelss computes the minimum norm solution to a complex linear least squares problem:

Minimize 2-norm(| b - A*x |).

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the first `min(m,n)` rows of A are overwritten with its right singular vectors, stored rowwise.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,M,N).
- **SING (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = SING(1)/SING(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `SING(i)` < = RCOND*SING(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **IRANK (output)**

The effective rank of A, i.e., the number of singular values which are greater than RCOND*SING(1).

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1, and also: LDWORK >= 2*min(M,N) + `max(M,N,NRHS)`
  For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **WORK2 (workspace)**
  `dimension(5*min(M,N))`

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

NAME

cgelsx - routine is deprecated and has been replaced by routine CGELSY

SYNOPSIS

```
 SUBROUTINE CGELSX( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND, IRANK,
*      WORK, WORK2, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER JPIVOT(*)
 REAL RCOND
 REAL WORK2(*)

 SUBROUTINE CGELSX_64( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND,
*      IRANK, WORK, WORK2, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER*8 JPIVOT(*)
 REAL RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSX( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [WORK2], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER, DIMENSION(:) :: JPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE GELSX_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [WORK2], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER(8), DIMENSION(:) :: JPIVOT
```

```
REAL :: RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgelsx(int m, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *jpivot, float rcond, int *irank, int *info);

void cgelsx_64(long m, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *jpivot, float rcond, long *irank, long *info);

PURPOSE

cgelsx routine is deprecated and has been replaced by routine CGELSY.

CGELSX computes the minimum-norm solution to a complex linear least squares problem:

```
    minimize || A * X - B ||
```

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by unitary transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X. If m >= n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements N+1:M in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is an initial column, otherwise it is a free column. Before the QR factorization of A, all initial columns are permuted to the leading positions; only the remaining free columns are moved as a result of column pivoting during the factorization. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **IRANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  (min(M,N) + max(N, 2*min(M,N)+NRHS)),
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgelsy - compute the minimum-norm solution to a complex linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE CGELSY( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, RWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER JPVT(*)
 REAL RCOND
 REAL RWORK(*)

 SUBROUTINE CGELSY_64( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, RWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 JPVT(*)
 REAL RCOND
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSY( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL :: RCOND
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE GELSY_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT,
*       RCOND, RANK, [WORK], [LWORK], [RWORK], [INFO])
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
```

```
INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
INTEGER(8), DIMENSION(:) :: JPVT
REAL :: RCOND
REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void cgelsy(int m, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *jpvt, float rcond, int *rank, int *info);

void cgelsy_64(long m, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *jpvt, float rcond, long *rank, long *info);

PURPOSE

cgelsy computes the minimum-norm solution to a complex linear least squares problem: minimize || A * X - B ||

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by unitary transformations from the right, arriving at the complete orthogonal factorization:

```
A * P = Q * [ T11 0 ] * Z

            [  0  0 ]
```

The minimum-norm solution is then

```
X = P * Z' [ inv(T11)*Q1'*B ]

           [        0       ]
```

where Q1 consists of the first RANK columns of Q.

This routine is basically identical to the original xGELSX except three differences:

```
o The permutation of matrix B (the right hand side) is faster and
  more simple.

o The call to the subroutine xGEQPF has been substituted by the
  the call to the subroutine xGEQP3. This subroutine is a Blas-3
  version of the QR factorization with column pivoting.
```

```
o Matrix B (the right hand side) is updated with Blas-3.
```

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPVT (input/output)**
  On entry, if `JPVT(i)` .ne. 0, the i-th column of A is permuted to the front of AP, otherwise column i is a free column. On exit, if `JPVT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **RANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. The unblocked strategy requires that: LWORK >= MN + MAX(2*MN, N+1, MN+NRHS) where MN = min(M,N). The block algorithm requires that: LWORK >= MN + MAX(2*MN, NB*(N+1), MN+MN*NB, MN+NB*NRHS) where NB is an upper bound on the blocksize returned by ILAENV for the routines CGEQP3, CTZRZF, CTZRQF, CUNMQR, and CUNMRZ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
E. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain

---

NAME

cgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

SYNOPSIS

```
 SUBROUTINE CGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, K, LDA, LDB, LDC

 SUBROUTINE CGEMM_64( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, K, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE GEMM( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A, [LDA],
*       B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, K, LDA, LDB, LDC

 SUBROUTINE GEMM_64( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A,
*       [LDA], B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, K, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void cgemm(char transa, char transb, int m, int n, int k, complex alpha, complex *a, int lda, complex *b, int ldb, complex beta, complex *c, int ldc);

void cgemm_64(char transa, char transb, long m, long n, long k, complex alpha, complex *a, long lda, complex *b, long ldb, complex beta, complex *c, long ldc);

PURPOSE

cgemm performs one of the matrix-matrix operations

C := alpha*op(A)*op(B) + beta*C

where op(X) is one of

`op(X)` = X or `op(X)` = X' or `op(X)` = conjg(X'), alpha and beta are scalars, and A, B and C are matrices, with `op(A)` an m by k matrix, `op(B)` a k by n matrix and C an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n', op(A) = A.

  TRANSA = 'T' or 't', op(A) = A'.

  TRANSA = 'C' or 'c', op(A) = conjg(A').

  Unchanged on exit.
- **TRANSB (input)**
  On entry, TRANSB specifies the form of op(B) to be used in the matrix multiplication as follows:

  TRANSB = 'N' or 'n', op(B) = B.

  TRANSB = 'T' or 't', op(B) = B'.

  TRANSB = 'C' or 'c', op(B) = conjg(B').

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix op(A) and of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **K (input)**
  On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K

> = 0. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  K when TRANSA = 'N' or 'n', and is M otherwise. Before entry with TRANSA = 'N' or 'n', the leading M by K part of the array A must contain the matrix A, otherwise the leading K by M part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA > = max(1, M), otherwise LDA > = max(1, K). Unchanged on exit.
- **B (input)**
  n when TRANSB = 'N' or 'n', and is k otherwise. Before entry with TRANSB = 'N' or 'n', the leading k by n part of the array B must contain the matrix B, otherwise the leading n by k part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSB = 'N' or 'n' then LDB > = max(1, k), otherwise LDB > = max(1, n). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n matrix (alpha*op(A)*op(B) + beta*C).
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

cgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE CGEMV( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER M, N, LDA, INCX, INCY

 SUBROUTINE CGEMV_64( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 M, N, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE GEMV( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, INCX, INCY

 SUBROUTINE GEMV_64( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void cgemv(char transa, int m, int n, complex alpha, complex *a, int lda, complex *x, int incx, complex beta, complex *y, int incy);

void cgemv_64(char transa, long m, long n, complex alpha, complex *a, long lda, complex *x, long incx, complex beta, complex *y, long incy);

PURPOSE

cgemv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*conjg(A')*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before

entry with BETA non-zero, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cgeqlf - compute a QL factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE CGEQLF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE CGEQLF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GEQLF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GEQLF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgeqlf(int m, int n, complex *a, int lda, complex *tau, int *info);

void cgeqlf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

cgeqlf computes a QL factorization of a complex M-by-N matrix A: A = Q * L.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m >= n, the lower triangle of the subarray `A(m-n+1:m,1:n)` contains the N-by-N lower triangular matrix L; if m <= n, the elements on and below the (n-m)-th superdiagonal contain the M-by-N lower trapezoidal matrix L; the remaining elements, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(m-k+i+1:m)` = 0 and `v(m-k+i)` = 1; `v(1:m-k+i-1)` is stored on exit in A(1:m-k+i-1,n-k+i), and tau in TAU(i).

NAME

cgeqp3 - compute a QR factorization with column pivoting of a matrix A

SYNOPSIS

```
 SUBROUTINE CGEQP3( M, N, A, LDA, JPVT, TAU, WORK, LWORK, RWORK,
*      INFO)
 COMPLEX A(LDA,*), TAU(*), WORK(*)
 INTEGER M, N, LDA, LWORK, INFO
 INTEGER JPVT(*)
 REAL RWORK(*)

 SUBROUTINE CGEQP3_64( M, N, A, LDA, JPVT, TAU, WORK, LWORK, RWORK,
*      INFO)
 COMPLEX A(LDA,*), TAU(*), WORK(*)
 INTEGER*8 M, N, LDA, LWORK, INFO
 INTEGER*8 JPVT(*)
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEQP3( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [RWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE GEQP3_64( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [RWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: JPVT
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void cgeqp3(int m, int n, complex *a, int lda, int *jpvt, complex *tau, int *info);

void cgeqp3_64(long m, long n, complex *a, long lda, long *jpvt, complex *tau, long *info);

PURPOSE

cgeqp3 computes a QR factorization with column pivoting of a matrix A: A*P = Q*R using Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper trapezoidal matrix R; the elements below the diagonal, together with the array TAU, represent the unitary matrix Q as a product of `min(M,N)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **JPVT (input/output)**
  On entry, if JPVT(J).ne.0, the J-th column of A is permuted to the front of A*P (a leading column); if `JPVT(J)` =0, the J-th column of A is a free column. On exit, if `JPVT(J)` =K, then the J-th column of A*P was the the K-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = N+1. For optimal performance LWORK > = (N+1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0: successful exit.
  ```

```
< 0: if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real/complex scalar, and v is a real/complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

Based on contributions by

```
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
X. Sun, Computer Science Dept., Duke University, USA
```

NAME

cgeqpf - routine is deprecated and has been replaced by routine CGEQP3

SYNOPSIS

```
SUBROUTINE CGEQPF( M, N, A, LDA, JPIVOT, TAU, WORK, WORK2, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, INFO
INTEGER JPIVOT(*)
REAL WORK2(*)

SUBROUTINE CGEQPF_64( M, N, A, LDA, JPIVOT, TAU, WORK, WORK2, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 JPIVOT(*)
REAL WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE GEQPF( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [WORK2],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: JPIVOT
REAL, DIMENSION(:) :: WORK2

SUBROUTINE GEQPF_64( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [WORK2],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: JPIVOT
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cgeqpf(int m, int n, complex *a, int lda, int *jpivot, complex *tau, int *info);

void cgeqpf_64(long m, long n, complex *a, long lda, long *jpivot, complex *tau, long *info);

PURPOSE

cgeqpf routine is deprecated and has been replaced by routine CGEQP3.

CGEQPF computes a QR factorization with column pivoting of a complex M-by-N matrix A: A*P = Q*R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper triangular matrix R; the elements below the diagonal, together with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is permuted to the front of A*P (a leading column); if `JPIVOT(i)` = 0, the i-th column of A is a free column. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n)
```

Each `H(i)` has the form

```
H  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i).

The matrix P is represented in jpvt as follows: If

```
jpvt(j)  = i
```

then the jth column of P is the ith canonical unit vector.

NAME

cgeqrf - compute a QR factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE CGEQRF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE CGEQRF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GEQRF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GEQRF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgeqrf(int m, int n, complex *a, int lda, complex *tau, int *info);

void cgeqrf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

cgeqrf computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and above the diagonal of the array contain the min(M,N)-by-N upper trapezoidal matrix R (R is upper triangular if m >= n); the elements below the diagonal, with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

NAME

cgerc - perform the rank 1 operation A := alpha*x*conjg(y') + A

SYNOPSIS

```
SUBROUTINE CGERC( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER M, N, INCX, INCY, LDA

SUBROUTINE CGERC_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 M, N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE GERC( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, INCX, INCY, LDA

SUBROUTINE GERC_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void cgerc(int m, int n, complex alpha, complex *x, int incx, complex *y, int incy, complex *a, int lda);

void cgerc_64(long m, long n, complex alpha, complex *x, long incx, complex *y, long incy, complex *a, long lda);

PURPOSE

cgerc performs the rank 1 operation A := alpha*x*conjg(y') + A where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

cgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CGERFS( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGERFS_64( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GERFS( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GERFS_64( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgerfs(char transa, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cgerfs_64(char transa, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cgerfs improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original N-by-N matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CGETRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgerqf - compute an RQ factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE CGERQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE CGERQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GERQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GERQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgerqf(int m, int n, complex *a, int lda, complex *tau, int *info);

void cgerqf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

cgerqf computes an RQ factorization of a complex M-by-N matrix A: A = R * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m <= n, the upper triangle of the subarray `A(1:m,n-m+1:n)` contains the M-by-M upper triangular matrix R; if m >= n, the elements on and above the (m-n)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1)' H(2)' . . . H(k)', where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `conjg(v(1:n-k+i-1))` is stored on exit in A(m-k+i,1:n-k+i-1), and tau in TAU(i).

NAME

cgeru - perform the rank 1 operation A := alpha*x*y' + A

SYNOPSIS

```
SUBROUTINE CGERU( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER M, N, INCX, INCY, LDA

SUBROUTINE CGERU_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 M, N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE GER( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, INCX, INCY, LDA

SUBROUTINE GER_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void cgeru(int m, int n, complex alpha, complex *x, int incx, complex *y, int incy, complex *a, int lda);

void cgeru_64(long m, long n, complex alpha, complex *x, long incx, complex *y, long incy, complex *a, long lda);

PURPOSE

cgeru performs the rank 1 operation A := alpha*x*y' + A where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

cgesdd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method

---

SYNOPSIS

```
 SUBROUTINE CGESDD( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, RWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER IWORK(*)
 REAL S(*), RWORK(*)

 SUBROUTINE CGESDD_64( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, RWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER*8 M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL S(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESDD( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT, [LDVT],
*       [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, U, VT
 INTEGER :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: S, RWORK

 SUBROUTINE GESDD_64( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT,
*       [LDVT], [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
```

```
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, U, VT
INTEGER(8) :: M, N, LDA, LDU, LDVT, LWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: S, RWORK
```

C INTERFACE

#include <sunperf.h>

void cgesdd(char jobz, int m, int n, complex *a, int lda, float *s, complex *u, int ldu, complex *vt, int ldvt, int *info);

void cgesdd_64(char jobz, long m, long n, complex *a, long lda, float *s, complex *u, long ldu, complex *vt, long ldvt, long *info);

PURPOSE

cgesdd computes the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method. The SVD is written = U * SIGMA * conjugate-transpose(V)

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M unitary matrix, and V is an N-by-N unitary matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns VT = V**H, not V.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U and all N rows of V**H are
returned in the arrays U and VT;
 = 'S':  the first min(M,N) columns of U and the first
min(M,N) rows of V**H are returned in the arrays U
and VT;
 = 'O':  If M  > = N, the first N columns of U are overwritten
on the array A and all rows of V**H are returned in
the array VT;
otherwise, all columns of U are returned in the
array U and the first M rows of V**H are overwritten
in the array VT;
 = 'N':  no columns of U or rows of V**H are computed.
```

- **M (input)**

The number of rows of the input matrix A. M >= 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBZ = 'O', A is overwritten with the first N columns of U (the left singular vectors, stored columnwise) if M >= N; A is overwritten with the first M rows of V**H (the right singular vectors, stored rowwise) otherwise. if JOBZ .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **S (output)**
  The singular values of A, sorted so that `S(i)` >= S(i+1).
- **U (output)**
  UCOL = M if JOBZ = 'A' or JOBZ = 'O' and M < N; UCOL = `min(M,N)` if JOBZ = 'S'. If JOBZ = 'A' or JOBZ = 'O' and M < N, U contains the M-by-M unitary matrix U; if JOBZ = 'S', U contains the first `min(M,N)` columns of U (the left singular vectors, stored columnwise); if JOBZ = 'O' and M >= N, or JOBZ = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBZ = 'S' or 'A' or JOBZ = 'O' and M < N, LDU >= M.
- **VT (output)**
  If JOBZ = 'A' or JOBZ = 'O' and M >= N, VT contains the N-by-N unitary matrix V**H; if JOBZ = 'S', VT contains the first `min(M,N)` rows of V**H (the right singular vectors, stored rowwise); if JOBZ = 'O' and M < N, or JOBZ = 'N', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBZ = 'A' or JOBZ = 'O' and M >= N, LDVT >= N; if JOBZ = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. if JOBZ = 'N', LWORK >= 2*min(M,N)+max(M,N). if JOBZ = 'O', LWORK >= 2*min(M,N)*min(M,N)+2*min(M,N)+max(M,N). if JOBZ = 'S' or 'A', LWORK >= min(M,N)*min(M,N)+2*min(M,N)+max(M,N). For good performance, LWORK should generally be larger. If LWORK < 0 but other input arguments are legal, `WORK(1)` returns optimal LWORK.
- **RWORK (workspace)**
  If JOBZ = 'N', LRWORK >= 7*min(M,N). Otherwise, LRWORK >= 5*min(M,N)*min(M,N) + 5*min(M,N)
- **IWORK (workspace)**
  `dimension(8*MIN(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.

> 0:  The updating process of SBDSDC did not converge.
```

FURTHER DETAILS

Based on contributions by

Ming Gu and Huan Ren, Computer Science Division, University of California at Berkeley, USA

NAME

cgesv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CGESV( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGESV_64( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GESV( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GESV_64( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgesv(int n, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void cgesv_64(long n, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cgesv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as

```
A = P * L * U,
```

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N coefficient matrix A. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution could not be computed.
```

NAME

cgesvd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors

---

SYNOPSIS

```
 SUBROUTINE CGESVD( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBU, JOBVT
 COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL SING(*), WORK2(*)

 SUBROUTINE CGESVD_64( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBU, JOBVT
 COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER*8 M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL SING(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVD( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, U, VT
 INTEGER :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL, DIMENSION(:) :: SING, WORK2

 SUBROUTINE GESVD_64( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, U, VT
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL, DIMENSION(:) :: SING, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgesvd(char jobu, char jobvt, int m, int n, complex *a, int lda, float *sing, complex *u, int ldu, complex *vt, int ldvt, int *info);

void cgesvd_64(char jobu, char jobvt, long m, long n, complex *a, long lda, float *sing, complex *u, long ldu, complex *vt, long ldvt, long *info);

PURPOSE

cgesvd computes the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors. The SVD is written = U * SIGMA * conjugate-transpose(V)

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M unitary matrix, and V is an N-by-N unitary matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns V**H, not V.

ARGUMENTS

- **JOBU (input)**
  Specifies options for computing all or part of the matrix U:

  ```
   = 'A':  all M columns of U are returned in array U:

   = 'S':  the first min(m,n) columns of U (the left singular
  vectors) are returned in the array U;
   = 'O':  the first min(m,n) columns of U (the left singular
  vectors) are overwritten on the array A;
   = 'N':  no columns of U (no left singular vectors) are
  computed.
  ```

- **JOBVT (input)**
  Specifies options for computing all or part of the matrix V**H:

  ```
   = 'A':  all N rows of V**H are returned in the array VT;

   = 'S':  the first min(m,n) rows of V**H (the right singular
  vectors) are returned in the array VT;
   = 'O':  the first min(m,n) rows of V**H (the right singular
  vectors) are overwritten on the array A;
   = 'N':  no rows of V**H (no right singular vectors) are
  computed.
  ```

  JOBVT and JOBU cannot both be 'O'.

- **M (input)**

The number of rows of the input matrix A. M >= 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBU = 'O', A is overwritten with the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBVT = 'O', A is overwritten with the first `min(m,n)` rows of V**H (the right singular vectors, stored rowwise); if JOBU .ne. 'O' and JOBVT .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **SING (output)**
  The singular values of A, sorted so that `SING(i)` >= SING(i+1).
- **U (output)**
  (LDU,M) if JOBU = 'A' or (LDU,min(M,N)) if JOBU = 'S'. If JOBU = 'A', U contains the M-by-M unitary matrix U; if JOBU = 'S', U contains the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBU = 'N' or 'O', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBU = 'S' or 'A', LDU >= M.
- **VT (output)**
  If JOBVT = 'A', VT contains the N-by-N unitary matrix V**H; if JOBVT = 'S', VT contains the first `min(m,n)` rows of V**H (the right singular vectors, stored rowwise); if JOBVT = 'N' or 'O', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBVT = 'A', LDVT >= N; if JOBVT = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1. LDWORK >= 2*MIN(M,N)+MAX(M,N) For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  DIMENSION(5*MIN(M,N)). On exit, if INFO > 0, `WORK2(1:MIN(M,N)-1)` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in SING (not necessarily sorted). B satisfies A = U * B * VT, so it has the same singular values as A, and singular vectors related by U and VT.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if CBDSQR did not converge, INFO specifies how many
superdiagonals of an intermediate bidiagonal form B
did not converge to zero. See the description of WORK2
above for details.
```

NAME

cgesvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CGESVX( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR, WORK,
*      WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGESVX_64( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF,
*      IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVX( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

```
 SUBROUTINE GESVX_64( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND,
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgesvx(char fact, char transa, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, char equed, float *rowsc, float *colsc, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cgesvx_64(char fact, char transa, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, char equed, float *rowsc, float *colsc, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cgesvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = P * L * U,

where P is a permutation matrix, L is a unit lower triangular
matrix, and U is upper triangular.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
   = 'N':  A * X  = B     (No transpose)

   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Conjugate transpose)
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the N-by-N matrix A. If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the factors L and U from the factorization A = P*L*U as computed by CGETRF. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).

- **IPIVOT (input)**

If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = P*L*U as computed by CGETRF; row i of the matrix was interchanged with row IPIVOT(i).

If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the original matrix A.

If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the equilibrated matrix A.

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**

dimension(2*N) On exit, WORK2(1) contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If WORK2(1) is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then WORK2(1) contains the reciprocal pivot growth factor for the leading INFO columns of A.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization has
been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

cgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE CGETF2( M, N, A, LDA, IPIV, INFO)
COMPLEX A(LDA,*)
INTEGER M, N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE CGETF2_64( M, N, A, LDA, IPIV, INFO)
COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE GETF2( [M], [N], A, [LDA], IPIV, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE GETF2_64( [M], [N], A, [LDA], IPIV, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void cgetf2(int m, int n, complex *a, int lda, int *ipiv, int *info);

void cgetf2_64(long m, long n, complex *a, long lda, long *ipiv, long *info);

PURPOSE

cgetf2 computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 2 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the m by n matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

cgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE CGETRF( M, N, A, LDA, IPIVOT, INFO)
COMPLEX A(LDA,*)
INTEGER M, N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGETRF_64( M, N, A, LDA, IPIVOT, INFO)
COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRF( [M], [N], A, [LDA], IPIVOT, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRF_64( [M], [N], A, [LDA], IPIVOT, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgetrf(int m, int n, complex *a, int lda, int *ipivot, int *info);

void cgetrf_64(long m, long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

cgetrf computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 3 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

cgetri - compute the inverse of a matrix using the LU factorization computed by CGETRF

SYNOPSIS

```
SUBROUTINE CGETRI( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGETRI_64( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRI( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRI_64( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgetri(int n, complex *a, int lda, int *ipivot, int *info);

void cgetri_64(long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

cgetri computes the inverse of a matrix using the LU factorization computed by CGETRF.

This method inverts U and then computes `inv(A)` by solving the system inv(A)*L = `inv(U)` for inv(A).

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the factors L and U from the factorization A = P*L*U as computed by CGETRF. On exit, if INFO = 0, the inverse of the original matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **WORK (workspace)**
  On exit, if INFO =0, then `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimal performance LDWORK > = N*NB, where NB is the optimal blocksize returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero; the matrix is
singular and its inverse could not be computed.
```

NAME

cgetrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF

SYNOPSIS

```
SUBROUTINE CGETRS( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGETRS_64( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRS( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: TRANSA
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRS_64( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
CHARACTER(LEN=1) :: TRANSA
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgetrs(char transa, int n, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void cgetrs_64(char transa, long n, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cgetrs solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

cggbak - form the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL

SYNOPSIS

```
 SUBROUTINE CGGBAK( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 COMPLEX V(LDV,*)
 INTEGER N, ILO, IHI, M, LDV, INFO
 REAL LSCALE(*), RSCALE(*)

 SUBROUTINE CGGBAK_64( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 COMPLEX V(LDV,*)
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 REAL LSCALE(*), RSCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAK( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M], V,
*       [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX, DIMENSION(:,:) :: V
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE

 SUBROUTINE GGBAK_64( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M],
*       V, [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX, DIMENSION(:,:) :: V
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE
```

C INTERFACE

#include <sunperf.h>

void cggbak(char job, char side, int n, int ilo, int ihi, float *lscale, float *rscale, int m, complex *v, int ldv, int *info);

void cggbak_64(char job, char side, long n, long ilo, long ihi, float *lscale, float *rscale, long m, complex *v, long ldv, long *info);

PURPOSE

cggbak forms the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required:

```
= 'N':  do nothing, return immediately;

= 'P':  do backward transformation for permutation only;

= 'S':  do backward transformation for scaling only;

= 'B':  do backward transformations for both permutation and
scaling.
JOB must be the same as the argument JOB supplied to CGGBAL.
```

- **SIDE (input)**

```
= 'R':  V contains right eigenvectors;

= 'L':  V contains left eigenvectors.
```

- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by CGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  The integers ILO and IHI determined by CGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **LSCALE (input)**
  Details of the permutations and/or scaling factors applied to the left side of A and B, as returned by CGGBAL.
- **RSCALE (input)**
  Details of the permutations and/or scaling factors applied to the right side of A and B, as returned by CGGBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**

On entry, the matrix of right or left eigenvectors to be transformed, as returned by CTGEVC. On exit, V is overwritten by the transformed eigenvectors.

- **LDV (input)**
  The leading dimension of the matrix V. LDV >= max(1,N).
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. Ward, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

cggbal - balance a pair of general complex matrices (A,B)

SYNOPSIS

```
 SUBROUTINE CGGBAL( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE, RSCALE,
*      WORK, INFO)
 CHARACTER * 1 JOB
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, LDA, LDB, ILO, IHI, INFO
 REAL LSCALE(*), RSCALE(*), WORK(*)

 SUBROUTINE CGGBAL_64( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE,
*      RSCALE, WORK, INFO)
 CHARACTER * 1 JOB
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, LDA, LDB, ILO, IHI, INFO
 REAL LSCALE(*), RSCALE(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAL( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, LDA, LDB, ILO, IHI, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE, WORK

 SUBROUTINE GGBAL_64( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, LDA, LDB, ILO, IHI, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE, WORK
```

C INTERFACE

#include <sunperf.h>

void cggbal(char job, int n, complex *a, int lda, complex *b, int ldb, int *ilo, int *ihi, float *lscale, float *rscale, int *info);

void cggbal_64(char job, long n, complex *a, long lda, complex *b, long ldb, long *ilo, long *ihi, float *lscale, float *rscale, long *info);

PURPOSE

cggbal balances a pair of general complex matrices (A,B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to ILO$-$1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem A*x = lambda*B*x.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A and B:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, LSCALE(I)  = 1.0
and RSCALE(I)  = 1.0 for i =1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the input matrix B. On exit, B is overwritten by the balanced matrix. If JOB = 'N', B is not referenced.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  ILO and IHI are set to integers such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N.

- **LSCALE (input)**
  Details of the permutations and scaling factors applied to the left side of A and B. If `P(j)` is the index of the row interchanged with row j, and `D(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (input)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `P(j)` is the index of the column interchanged with column j, and `D(j)` is the scaling factor applied to column j, then `RSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. WARD, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

cgges - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR)

SYNOPSIS

```
 SUBROUTINE CGGES( JOBVSL, JOBVSR, SORT, SELCTG, N, A, LDA, B, LDB,
*      SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK, RWORK,
*      BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL SELCTG
 LOGICAL BWORK(*)
 REAL RWORK(*)

 SUBROUTINE CGGES_64( JOBVSL, JOBVSR, SORT, SELCTG, N, A, LDA, B,
*      LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK,
*      RWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL*8 SELCTG
 LOGICAL*8 BWORK(*)
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGES( JOBVSL, JOBVSR, SORT, SELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK],
*       [LWORK], [RWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL :: SELCTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: RWORK
```

```
 SUBROUTINE GGES_64( JOBVSL, JOBVSR, SORT, SELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK],
*       [LWORK], [RWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL(8) :: SELCTG
 LOGICAL(8), DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void cgges(char jobvsl, char jobvsr, char sort, logical(*selctg)(complex,complex), int n, complex *a, int lda, complex *b, int ldb, int *sdim, complex *alpha, complex *beta, complex *vsl, int ldvsl, complex *vsr, int ldvsr, int *info);

void cgges_64(char jobvsl, char jobvsr, char sort, logical(*selctg)(complex,complex), long n, complex *a, long lda, complex *b, long ldb, long *sdim, complex *alpha, complex *beta, complex *vsl, long ldvsl, complex *vsr, long ldvsr, long *info);

PURPOSE

cgges computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR). This gives the generalized Schur factorization

```
(A,B) = ( (VSL)*S*(VSR)**H, (VSL)*T*(VSR)**H )
```

where (VSR)**H is the conjugate-transpose of VSR.

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper triangular matrix S and the upper triangular matrix T. The leading columns of VSL and VSR then form an unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver CGGEV instead, which is faster.)

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal elements of T are non-negative real numbers.

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

  ```
  = 'S':  Eigenvalues are ordered (see SELCTG).
  ```

- **SELCTG (input)**
  SELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', SELCTG is not referenced. If SORT = 'S', SELCTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue `ALPHA(j)/BETA(j)` is selected if `SELCTG(ALPHA(j),BETA(j))` is true.

  Note that a selected complex eigenvalue may no longer satisfy `SELCTG(ALPHA(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+2 (See INFO below).

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.

- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).

- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which SELCTG is true.

- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHA(j), j =1,...,N and BETA(j), j =1,...,N are the diagonals of the complex Schur form (A,B) output by CGGES. The `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).

- **BETA (output)**
  See description of ALPHA.

- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.

- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > = 1, and if JOBVSL = 'V', LDVSL > = N.

- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,2*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(8*N)`
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in CHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy SELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering falied in CTGSEN.
```

NAME

cggesx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T),

---

SYNOPSIS

```
 SUBROUTINE CGGESX( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, N, A, LDA,
*      B, LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, RCONDE,
*      RCONDV, WORK, LWORK, RWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELCTG
 LOGICAL BWORK(*)
 REAL RCONDE(*), RCONDV(*), RWORK(*)

 SUBROUTINE CGGESX_64( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, N, A,
*      LDA, B, LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, RCONDE,
*      RCONDV, WORK, LWORK, RWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELCTG
 LOGICAL*8 BWORK(*)
 REAL RCONDE(*), RCONDV(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGESX( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, [N], A, [LDA],
*       B, [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], RCONDE,
*       RCONDV, [WORK], [LWORK], [RWORK], [IWORK], [LIWORK], [BWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
```

```
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL :: SELCTG
  LOGICAL, DIMENSION(:) :: BWORK
  REAL, DIMENSION(:) :: RCONDE, RCONDV, RWORK

  SUBROUTINE GGESX_64( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, [N], A,
 *       [LDA], B, [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR],
 *       RCONDE, RCONDV, [WORK], [LWORK], [RWORK], [IWORK], [LIWORK],
 *       [BWORK], [INFO])
  CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
  COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
  COMPLEX, DIMENSION(:,:) :: A, B, VSL, VSR
  INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8) :: SELCTG
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL, DIMENSION(:) :: RCONDE, RCONDV, RWORK
```

C INTERFACE

#include <sunperf.h>

void cggesx(char jobvsl, char jobvsr, char sort, logical(*selctg)(complex,complex), char sense, int n, complex *a, int lda, complex *b, int ldb, int *sdim, complex *alpha, complex *beta, complex *vsl, int ldvsl, complex *vsr, int ldvsr, float *rconde, float *rcondv, int *info);

void cggesx_64(char jobvsl, char jobvsr, char sort, logical(*selctg)(complex,complex), char sense, long n, complex *a, long lda, complex *b, long ldb, long *sdim, complex *alpha, complex *beta, complex *vsl, long ldvsl, complex *vsr, long ldvsr, float *rconde, float *rcondv, long *info);

PURPOSE

cggesx computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T), and, optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization A,B) = ((VSL) S (VSR)**H, (VSL) T (VSR)**H)

where (VSR)**H is the conjugate-transpose of VSR.

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper triangular matrix S and the upper triangular matrix T; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (RCONDV). The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or for both being zero.

A pair of matrices (S,T) is in generalized complex Schur form if T is upper triangular with non-negative diagonal and S is upper triangular.

ARGUMENTS

- **JOBVSL (input)**

```
= 'N':  do not compute the left Schur vectors;

= 'V':  compute the left Schur vectors.
```

- **JOBVSR (input)**

```
= 'N':  do not compute the right Schur vectors;

= 'V':  compute the right Schur vectors.
```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S':  Eigenvalues are ordered (see SELCTG).
```

- **SELCTG (input)**
  SELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', SELCTG is not referenced. If SORT = 'S', SELCTG is used to select eigenvalues to sort to the top left of the Schur form. Note that a selected complex eigenvalue may no longer satisfy `SELCTG(ALPHA(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+3 see INFO below).
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N' : None are computed;

```
= 'E' : Computed for average of selected eigenvalues only;

= 'V' : Computed for selected deflating subspaces only;

= 'B' : Computed for both.
If SENSE  = 'E', 'V', or 'B', SORT must equal 'S'.
```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which SELCTG is true.
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHA(j)` and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T). `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).

- **BETA (output)**
  See description of ALPHA.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >=1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', `RCONDE(1)` and `RCONDE(2)` contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', `RCONDV(1)` and `RCONDV(2)` contain the reciprocal condition number for the selected deflating subspaces. Not referenced if SENSE = 'N' or 'E'.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 2*N. If SENSE = 'E', 'V', or 'B', LWORK >= MAX(2*N, 2*SDIM*(N-SDIM)).
- **RWORK (workspace)**
  `dimension(8*N)` Real workspace.
- **IWORK (workspace)**
  Not referenced if SENSE = 'N'. On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array WORK. LIWORK >= N+2.
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in CHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy SELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering failed in CTGSEN.
```

NAME

cggev - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

---

SYNOPSIS

```
 SUBROUTINE CGGEV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA, VL,
*      LDVL, VR, LDVR, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL RWORK(*)

 SUBROUTINE CGGEV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VL, LDVL, VR, LDVR, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGEV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA, BETA,
*       VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE GGEV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void cggev(char jobvl, char jobvr, int n, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *vl, int ldvl, complex *vr, int ldvr, int *info);

void cggev_64(char jobvl, char jobvr, long n, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *vl, long ldvl, complex *vr, long ldvr, long *info);

PURPOSE

cggev computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right generalized eigenvector `v(j)` corresponding to the generalized eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j).
```

The left generalized eigenvector `u(j)` corresponding to the generalized eigenvalues `lambda(j)` of (A,B) satisfies

```
u(j)**H * A = lambda(j) * u(j)**H * B
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten.
- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See description of ALPHA.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,2*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and BETA(j) should be
correct for j =INFO+1,...,N.
 > N:   =N+1: other then QZ iteration failed in SHGEQZ,

 =N+2: error return from STGEVC.
```

NAME

cggevx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

---

SYNOPSIS

```
 SUBROUTINE CGGEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B, LDB,
*      ALPHA, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE, RSCALE, ABNRM,
*      BBNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, IWORK, BWORK, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL BWORK(*)
 REAL ABNRM, BBNRM
 REAL LSCALE(*), RSCALE(*), RCONDE(*), RCONDV(*), RWORK(*)

 SUBROUTINE CGGEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B,
*      LDB, ALPHA, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE, RSCALE,
*      ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, IWORK, BWORK,
*      INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 BWORK(*)
 REAL ABNRM, BBNRM
 REAL LSCALE(*), RSCALE(*), RCONDE(*), RCONDV(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
*       [LDB], ALPHA, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI, LSCALE,
*       RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK], [RWORK],
*       [IWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
```

```
  COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
  INTEGER :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL, DIMENSION(:) :: BWORK
  REAL :: ABNRM, BBNRM
  REAL, DIMENSION(:) :: LSCALE, RSCALE, RCONDE, RCONDV, RWORK

  SUBROUTINE GGEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHA, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI, LSCALE,
 *       RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK], [RWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
  COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
  INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL :: ABNRM, BBNRM
  REAL, DIMENSION(:) :: LSCALE, RSCALE, RCONDE, RCONDV, RWORK
```

C INTERFACE

#include <sunperf.h>

void cggevx(char balanc, char jobvl, char jobvr, char sense, int n, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *vl, int ldvl, complex *vr, int ldvr, int *ilo, int *ihi, float *lscale, float *rscale, float *abnrm, float *bbnrm, float *rconde, float *rcondv, int *info);

void cggevx_64(char balanc, char jobvl, char jobvr, char sense, long n, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *vl, long ldvl, complex *vr, long ldvr, long *ilo, long *ihi, float *lscale, float *rscale, float *abnrm, float *bbnrm, float *rconde, float *rcondv, long *info);

PURPOSE

cggevx computes for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally, it also computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, LSCALE, RSCALE, ABNRM, and BBNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right eigenvectors (RCONDV).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j) .
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B.
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **BALANC (input)**
  Specifies the balance option to be performed:

```
  = 'N':  do not diagonally scale or permute;

  = 'P':  permute only;

  = 'S':  scale only;

  = 'B':  both permute and scale.
Computed reciprocal condition numbers will be for the
matrices after permuting and/or balancing. Permuting does
not change condition numbers (in exact arithmetic), but
balancing does.
```

- **JOBVL (input)**

```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
```

- **JOBVR (input)**

```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
```

- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': none are computed;

```
  = 'E': computed for eigenvalues only;

  = 'V': computed for eigenvectors only;

  = 'B': computed for eigenvalues and eigenvectors.
```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then A contains the first part of the complex Schur form of the ``balanced'' versions of the input A and B.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then B contains the second part of the complex Schur form of the ``balanced'' versions of the input A and B.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues.

Note: the quotient ALPHA(j)/BETA(j)) may easily over- or underflow, and BETA(j) may even be zero. Thus, the user should avoid naively computing the ratio ALPHA/BETA. However, ALPHA will be always less than and usually comparable with norm(A) in magnitude, and BETA always less than and usually comparable with norm(B).

- **BETA (output)**
  See description of ALPHA.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors u(j) are stored one after another in the columns of VL, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors v(j) are stored one after another in the columns of VR, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO is an integer value such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  IHI is an integer value such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **LSCALE (output)**
  Details of the permutations and scaling factors applied to the left side of A and B. If PL(j) is the index of the row interchanged with row j, and DL(j) is the scaling factor applied to row j, then LSCALE(j) = PL(j) for j = 1,...,ILO-1 = DL(j) for j = ILO,...,IHI = PL(j) for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (output)**
  Details of the permutations and scaling factors applied to the right side of A and B. If PR(j) is the index of the column interchanged with column j, and DR(j) is the scaling factor applied to column j, then RSCALE(j) = PR(j) for j = 1,...,ILO-1 = DR(j) for j = ILO,...,IHI = PR(j) for j = IHI+1,...,N The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix A.
- **BBNRM (output)**
  The one-norm of the balanced matrix B.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If SENSE = 'V', RCONDE is not referenced.
- **RCONDV (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute RCONDV(j), RCONDV(j) is set to 0; this can only occur when the true value would be very small anyway. If SENSE = 'E', RCONDV is not referenced. Not referenced if JOB = 'E'.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,2*N). If SENSE = 'N' or 'E', LWORK >= 2*N. If SENSE = 'V' or 'B', LWORK >= 2*N*N+2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK

array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **RWORK (workspace)**
  `dimension(6*N)` Real workspace.
- **IWORK (workspace)**
  `dimension(N+2)` If SENSE = 'E', IWORK is not referenced.
- **BWORK (workspace)**
  `dimension(N)` If SENSE = 'N', BWORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and BETA(j) should be correct
for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in CHGEQZ.

 =N+2: error return from CTGEVC.
```

FURTHER DETAILS

Balancing a matrix pair (A,B) includes, first, permuting rows and columns to isolate eigenvalues, second, applying diagonal similarity transformation to the rows and columns to make the rows and columns as close in norm as possible. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.11.1.2 of LAPACK Users' Guide.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(ABNRM, BBNRM) / `RCONDE(I)`

An approximate error bound for the angle between the i-th computed eigenvector `VL(i)` or `VR(i)` is given by

PS * norm(ABNRM, BBNRM) / DIF(i).

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see section 4.11 of LAPACK User's Guide.

NAME

cggglm - solve a general Gauss-Markov linear model (GLM) problem

---

SYNOPSIS

```
 SUBROUTINE CGGGLM( N, M, P, A, LDA, B, LDB, D, X, Y, WORK, LDWORK,
*      INFO)
 COMPLEX A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
 INTEGER N, M, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE CGGGLM_64( N, M, P, A, LDA, B, LDB, D, X, Y, WORK,
*      LDWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
 INTEGER*8 N, M, P, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGGLM( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y, [WORK],
*       [LDWORK], [INFO])
 COMPLEX, DIMENSION(:) :: D, X, Y, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, M, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE GGGLM_64( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y,
*       [WORK], [LDWORK], [INFO])
 COMPLEX, DIMENSION(:) :: D, X, Y, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, M, P, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cggglm(int n, int m, int p, complex *a, int lda, complex *b, int ldb, complex *d, complex *x, complex *y, int *info);

void cggglm_64(long n, long m, long p, complex *a, long lda, complex *b, long ldb, complex *d, complex *x, complex *y,

long *info);

PURPOSE

cggglm solves a general Gauss-Markov linear model (GLM) problem:

```
        minimize || y ||_2   subject to   d = A*x + B*y

            x
```

where A is an N-by-M matrix, B is an N-by-P matrix, and d is a given N-vector. It is assumed that M <= N <= M+P, and

```
           rank(A) = M    and    rank( A B ) = N.
```

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of A and B.

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

```
             minimize || inv(B)*(d-A*x) ||_2

                 x
```

where `inv(B)` denotes the inverse of B.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. 0 < = M < = N.
- **P (input)**
  The number of columns of the matrix B. P > = N-M.
- **A (input/output)**
  On entry, the N-by-M matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-P matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **D (input/output)**
  On entry, D is the left hand side of the GLM equation. On exit, D is destroyed.
- **X (output)**
  On exit, X and Y are the solutions of the GLM problem.
- **Y (output)**

On exit, X and Y are the solutions of the GLM problem.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N+M+P). For optimum performance, LDWORK >= M+min(N,P)+max(N,P)*NB, where NB is an upper bound for the optimal blocksizes for CGEQRF, CGERQF, CUNMQR and CUNMRQ.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

cgghrd - reduce a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular

---

SYNOPSIS

```
 SUBROUTINE CGGHRD( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO

 SUBROUTINE CGGHRD_64( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGHRD( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO

 SUBROUTINE GGHRD_64( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
```

C INTERFACE

#include <sunperf.h>

void cgghrd(char compq, char compz, int n, int ilo, int ihi, complex *a, int lda, complex *b, int ldb, complex *q, int ldq, complex *z, int ldz, int *info);

void cgghrd_64(char compq, char compz, long n, long ilo, long ihi, complex *a, long lda, complex *b, long ldb, complex *q, long ldq, complex *z, long ldz, long *info);

PURPOSE

cgghrd reduces a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular: Q' * A * Z = H and Q' * B * Z = T, where H is upper Hessenberg, T is upper triangular, and Q and Z are unitary, and ' means conjugate transpose.

The unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q1 and Z1, so that

1 * A * Z1' = (Q1*Q) * H * (Z1*Z)' 1 * B * Z1' = (Q1*Q) * T * (Z1*Z)'

ARGUMENTS

- **COMPQ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'V': Q must contain a unitary matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **COMPZ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'V': Q must contain a unitary matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGGBAL; otherwise they should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See description of ILO.
- **A (input/output)**

On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the rest is set to zero.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. On exit, the upper triangular matrix T = Q' B Z. The elements below the diagonal are set to zero.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **Q (input/output)**
  If COMPQ ='N': Q is not referenced.

  If COMPQ ='I': on entry, Q need not be set, and on exit it contains the unitary matrix Q, where Q' is the product of the Givens transformations which are applied to A and B on the left. If COMPQ ='V': on entry, Q must contain a unitary matrix Q1, and on exit this is overwritten by Q1*Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= N if COMPQ ='V' or 'I'; LDQ >= 1 otherwise.
- **Z (input/output)**
  If COMPZ ='N': Z is not referenced.

  If COMPZ ='I': on entry, Z need not be set, and on exit it contains the unitary matrix Z, which is the product of the Givens transformations which are applied to A and B on the right. If COMPZ ='V': on entry, Z must contain a unitary matrix Z1, and on exit this is overwritten by Z1*Z.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= N if COMPZ ='V' or 'I'; LDZ >= 1 otherwise.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

This routine reduces A to Hessenberg and B to triangular form by an unblocked reduction, as described in _Matrix_Computations_, by Golub and van Loan (Johns Hopkins Press).

NAME

cgglse - solve the linear equality-constrained least squares (LSE) problem

---

SYNOPSIS

```
 SUBROUTINE CGGLSE( M, N, P, A, LDA, B, LDB, C, D, X, WORK, LDWORK,
*      INFO)
 COMPLEX A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
 INTEGER M, N, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE CGGLSE_64( M, N, P, A, LDA, B, LDB, C, D, X, WORK,
*      LDWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
 INTEGER*8 M, N, P, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGLSE( [M], [N], [P], A, [LDA], B, [LDB], C, D, X, [WORK],
*       [LDWORK], [INFO])
 COMPLEX, DIMENSION(:) :: C, D, X, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE GGLSE_64( [M], [N], [P], A, [LDA], B, [LDB], C, D, X,
*       [WORK], [LDWORK], [INFO])
 COMPLEX, DIMENSION(:) :: C, D, X, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, P, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cgglse(int m, int n, int p, complex *a, int lda, complex *b, int ldb, complex *c, complex *d, complex *x, int *info);

void cgglse_64(long m, long n, long p, complex *a, long lda, complex *b, long ldb, complex *c, complex *d, complex *x,

long *info);

PURPOSE

cgglse solves the linear equality-constrained least squares (LSE) problem:

```
minimize || c - A*x ||_2   subject to   B*x = d
```

where A is an M-by-N matrix, B is a P-by-N matrix, c is a given M-vector, and d is a given P-vector. It is assumed that

P <= N <= M+P, and

```
rank(B) = P and  rank( ( A ) ) = N.
                       ( ( B ) )
```

These conditions ensure that the LSE problem has a unique solution, which is obtained using a GRQ factorization of the matrices B and A.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **P (input)**
  The number of rows of the matrix B. 0 < = P < = N < = M+P.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **C (input/output)**
  On entry, C contains the right hand side vector for the least squares part of the LSE problem. On exit, the residual sum of squares for the solution is given by the sum of squares of elements N-P+1 to M of vector C.
- **D (input/output)**
  On entry, D contains the right hand side vector for the constrained equation. On exit, D is destroyed.
- **X (output)**
  On exit, X is the solution of the LSE problem.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,M+N+P). For optimum performance LDWORK > = P+min(M,N)+max(M,N)*NB, where NB is an upper bound for the optimal blocksizes for CGEQRF, CGERQF,

CUNMQR and CUNMRQ.

If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

cggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

SYNOPSIS

```
 SUBROUTINE CGGQRF( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER N, M, P, LDA, LDB, LWORK, INFO

 SUBROUTINE CGGQRF_64( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER*8 N, M, P, LDA, LDB, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGQRF( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, M, P, LDA, LDB, LWORK, INFO

 SUBROUTINE GGQRF_64( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, M, P, LDA, LDB, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cggqrf(int n, int m, int p, complex *a, int lda, complex *taua, complex *b, int ldb, complex *taub, int *info);

void cggqrf_64(long n, long m, long p, complex *a, long lda, complex *taua, complex *b, long ldb, complex *taub, long *info);

PURPOSE

cggqrf computes a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B:

```
A = Q*R,        B = Q*T*Z,
```

where Q is an N-by-N unitary matrix, Z is a P-by-P unitary matrix, and R and T assume one of the forms:

if N >= M, R = (R11) M , or if N < M, R = (R11 R12) N, (0) N-M N M-N M

where R11 is upper triangular, and

if N <= P, T = (0 T12) N, or if N > P, T = (T11) N-P, P-N N (T21) P P

where T12 or T21 is upper triangular.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of inv(B)*A:

```
inv(B)*A = Z'*(inv(T)*R)
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the conjugate transpose of matrix Z.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. M > = 0.
- **P (input)**
  The number of columns of the matrix B. P > = 0.
- **A (input)**
  On entry, the N-by-M matrix A. On exit, the elements on and above the diagonal of the array contain the min(N,M)-by-M upper trapezoidal matrix R (R is upper triangular if N > = M); the elements below the diagonal, with the array TAUA, represent the unitary matrix Q as a product of `min(N,M)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q (see Further Details).
- **B (input)**
  On entry, the N-by-P matrix B. On exit, if N < = P, the upper triangle of the subarray `B(1:N,P-N+1:P)` contains the N-by-N upper triangular matrix T; if N > P, the elements on and above the (N-P)-th subdiagonal contain the N-by-P upper trapezoidal matrix T; the remaining elements, with the array TAUB, represent the unitary matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the QR factorization of an N-by-M matrix, NB2 is the optimal blocksize for the RQ factorization of an N-by-P matrix, and NB3 is the optimal blocksize for a call of CUNMQR.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(n,m).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i+1:n,i), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine CUNGQR.

To use Q to update another matrix, use LAPACK subroutine CUNMQR.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(n,p).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a complex scalar, and v is a complex vector with `v(p-k+i+1:p)` = 0 and `v(p-k+i)` = 1; `v(1:p-k+i-1)` is stored on exit in B(n-k+i,1:p-k+i-1), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine CUNGRQ.

To use Z to update another matrix, use LAPACK subroutine CUNMRQ.

NAME

cggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

SYNOPSIS

```
 SUBROUTINE CGGRQF( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER M, P, N, LDA, LDB, LWORK, INFO

 SUBROUTINE CGGRQF_64( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER*8 M, P, N, LDA, LDB, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGRQF( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, P, N, LDA, LDB, LWORK, INFO

 SUBROUTINE GGRQF_64( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX, DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, P, N, LDA, LDB, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cggrqf(int m, int p, int n, complex *a, int lda, complex *taua, complex *b, int ldb, complex *taub, int *info);

void cggrqf_64(long m, long p, long n, complex *a, long lda, complex *taua, complex *b, long ldb, complex *taub, long *info);

PURPOSE

cggrqf computes a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B:

```
A = R*Q,        B = Z*T*Q,
```

where Q is an N-by-N unitary matrix, Z is a P-by-P unitary matrix, and R and T assume one of the forms:

if M <= N, R = (0 R12) M, or if M > N, R = (R11) M-N, N-M M (R21) N N

where R12 or R21 is upper triangular, and

if P >= N, T = (T11) N , or if P < N, T = (T11 T12) P, (0) P-N P N-P N

where T11 is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of A*inv(B):

```
A*inv(B) = (R*inv(T))*Z'
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the conjugate transpose of the matrix Z.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **P (input)**
  The number of rows of the matrix B. P > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M < = N, the upper triangle of the subarray `A(1:M,N-M+1:N)` contains the M-by-M upper triangular matrix R; if M > N, the elements on and above the (M-N)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAUA, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q (see Further Details).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, the elements on and above the diagonal of the array contain the min(P,N)-by-N upper trapezoidal matrix T (T is upper triangular if P > = N); the elements below the diagonal, with the array TAUB, represent the unitary matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the RQ factorization of an M-by-N matrix, NB2 is the optimal blocksize for the QR factorization of a P-by-N matrix, and NB3 is the optimal blocksize for a call of CUNMRQ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO =-i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a complex scalar, and v is a complex vector with `v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine CUNGRQ.

To use Q to update another matrix, use LAPACK subroutine CUNMRQ.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(p,n).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:p)` is stored on exit in B(i+1:p,i), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine CUNGQR.

To use Z to update another matrix, use LAPACK subroutine CUNMQR.

NAME

cggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B

SYNOPSIS

```
 SUBROUTINE CGGSVD( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B, LDB,
*      ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, WORK2, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER IWORK3(*)
 REAL ALPHA(*), BETA(*), WORK2(*)

 SUBROUTINE CGGSVD_64( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B,
*      LDB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, WORK2, IWORK3,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER*8 M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK3(*)
 REAL ALPHA(*), BETA(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVD( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       [WORK2], IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK3
 REAL, DIMENSION(:) :: ALPHA, BETA, WORK2

 SUBROUTINE GGSVD_64( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       [WORK2], IWORK3, [INFO])
```

```
CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
INTEGER(8) :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
INTEGER(8), DIMENSION(:) :: IWORK3
REAL, DIMENSION(:) :: ALPHA, BETA, WORK2
```

C INTERFACE

#include <sunperf.h>

void cggsvd(char jobu, char jobv, char jobq, int m, int n, int p, int *k, int *l, complex *a, int lda, complex *b, int ldb, float *alpha, float *beta, complex *u, int ldu, complex *v, int ldv, complex *q, int ldq, int *iwork3, int *info);

void cggsvd_64(char jobu, char jobv, char jobq, long m, long n, long p, long *k, long *l, complex *a, long lda, complex *b, long ldb, float *alpha, float *beta, complex *u, long ldu, complex *v, long ldv, complex *q, long ldq, long *iwork3, long *info);

PURPOSE

cggsvd computes the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B:

```
      U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R )
```

where U, V and Q are unitary matrices, and Z' means the conjugate transpose of Z. Let K+L = the effective numerical rank of the matrix (A',B')', then R is a (K+L)-by-(K+L) nonsingular upper triangular matrix, D1 and D2 are M-by-(K+L) and P-by-(K+L) ``diagonal'' matrices and of the following structures, respectively:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
            M-K-L ( 0  0 )

                  K  L
       D2 =   L ( 0  S )
            P-L ( 0  0 )

                N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 )
            L (  0    0   R22 )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),

  S = diag( BETA(K+1),  ... , BETA(K+L) ),

  C**2 + S**2 = I.

  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                  K M-K K+L-M

       D1 =   K ( I  0    0   )

            M-K ( 0  C    0   )

                    K M-K K+L-M

       D2 =   M-K ( 0  S    0  )

            K+L-M ( 0  0    I  )

              P-L ( 0  0    0  )

                   N-K-L  K   M-K  K+L-M

  ( 0 R ) =     K ( 0    R11  R12  R13  )

              M-K ( 0     0   R22  R23  )

            K+L-M ( 0     0    0   R33  )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(M) ),

  S = diag( BETA(K+1),  ... , BETA(M) ),

  C**2 + S**2 = I.

  (R11 R12 R13 ) is stored in A(1:M, N-K-L+1:N), and R33 is stored
  ( 0  R22 R23 )

  in B(M-K+1:L,N+M-K-L+1:N) on exit.
```

The routine computes C, S, R, and optionally the unitary

transformation matrices U, V and Q.

In particular, if B is an N-by-N nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A*inv(B):

```
A*inv(B) = U*(D1*inv(D2))*V'.
```

If (A',B')' has orthnormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem: A'*A x = lambda* B'*B x.

In some literature, the GSVD of A and B is presented in the form U'*A*X = (0 D1), V'*B*X = (0 D2)

where U and V are orthogonal and X is nonsingular, and D1 and D2 are ``diagonal''. The former GSVD form can be converted to the latter form by taking the nonsingular matrix X as

```
X = Q*(  I   0    )

       (  0 inv(R) )
```

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Unitary matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Unitary matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Unitary matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.

- **N (input)**
  The number of columns of the matrices A and B. N > = 0.

- **P (input)**
  The number of rows of the matrix B. P > = 0.

- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.

- **L (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.

- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular matrix R, or part of R. See Purpose for details.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).

- **B (input/output)**

On entry, the P-by-N matrix B. On exit, B contains part of the triangular matrix R if M-K-L < 0. See Purpose for details.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1, ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L > = 0, ALPHA(K+1:K+L) = C,

  BETA(K+1:K+L) = S, or if M-K-L < 0, ALPHA(K+1:M) = C, ALPHA(M+1:K+L) = 0

  BETA(K+1:M) = S, BETA(M+1:K+L) = 1 and ALPHA(K+L+1:N) = 0

  BETA(K+L+1:N) = 0
- **BETA (output)**
  See description of ALPHA.
- **U (output)**
  If JOBU = 'U', U contains the M-by-M unitary matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU > = max(1,M) if JOBU = 'U'; LDU > = 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the P-by-P unitary matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,P) if JOBV = 'V'; LDV > = 1 otherwise.
- **Q (output)**
  If JOBQ = 'Q', Q contains the N-by-N unitary matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N) if JOBQ = 'Q'; LDQ > = 1 otherwise.
- **WORK (workspace)**
  dimension(MAX(3*N,M,P)+N)
- **WORK2 (workspace)**
  dimension(2*N)
- **IWORK3 (output)**
  dimension(N) On exit, IWORK3 stores the sorting information. More precisely, the following loop will sort ALPHA for I = K+1, min(M,K+L) swap ALPHA(I) and ALPHA(IWORK3(I)) endfor such that ALPHA(1) > = ALPHA(2) > = ... > = ALPHA(N).
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = 1, the Jacobi-type procedure failed to
converge.  For further details, see subroutine CTGSJA.
```

NAME

cggsvp - compute unitary matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

SYNOPSIS

```
 SUBROUTINE CGGSVP( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB, TOLA,
*      TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, RWORK, TAU, WORK,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
 INTEGER M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER IWORK(*)
 REAL TOLA, TOLB
 REAL RWORK(*)

 SUBROUTINE CGGSVP_64( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB,
*      TOLA, TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, RWORK, TAU,
*      WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
 INTEGER*8 M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK(*)
 REAL TOLA, TOLB
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVP( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B, [LDB],
*       TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK], [RWORK],
*       [TAU], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: RWORK
```

```
 SUBROUTINE GGSVP_64( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B,
*       [LDB], TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK],
*       [RWORK], [TAU], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER(8) :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void cggsvp(char jobu, char jobv, char jobq, int m, int p, int n, complex *a, int lda, complex *b, int ldb, float tola, float tolb, int *k, int *l, complex *u, int ldu, complex *v, int ldv, complex *q, int ldq, int *info);

void cggsvp_64(char jobu, char jobv, char jobq, long m, long p, long n, complex *a, long lda, complex *b, long ldb, float tola, float tolb, long *k, long *l, complex *u, long ldu, complex *v, long ldv, complex *q, long ldq, long *info);

PURPOSE

cggsvp computes unitary matrices U, V and Q such that L (0 0 A23)

```
          M-K-L ( 0     0    0  )

                 N-K-L  K    L

        =     K ( 0    A12  A13 )  if M-K-L < 0;

            M-K ( 0     0   A23 )

                 N-K-L  K    L

 V'*B*Q =   L ( 0     0   B13 )

          P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal. K+L = the effective numerical rank of the (M+P)-by-N matrix (A',B')'. Z' denotes the conjugate transpose of Z.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine CGGSVD.

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Unitary matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Unitary matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Unitary matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular (or trapezoidal) matrix described in the Purpose section.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix described in the Purpose section.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the thresholds to determine the effective numerical rank of matrix B and a subblock of A. Generally, they are set to TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS. The size of TOLA and TOLB may affect the size of backward errors of the decomposition.
- **TOLB (input)**
  See description of TOLA.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose section. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **U (output)**
  If JOBU = 'U', U contains the unitary matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the unitary matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**

The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (output)**
  If JOBQ = 'Q', Q contains the unitary matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **IWORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(2*N)`
- **TAU (workspace)**
  `dimension(N)`
- **WORK (workspace)**
  `dimension(MAX(3*N,M,P))`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The subroutine uses LAPACK subroutine CGEQPF for the QR factorization with column pivoting to detect the effective numerical rank of the a matrix. It may be replaced by a better rank determination strategy.

NAME

cgssco - General sparse solver condition number estimate.

SYNOPSIS

```
SUBROUTINE CGSSCO ( COND, HANDLE, IER )

INTEGER           IER
REAL              COND
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSCO - Condition number estimate.

PARAMETERS

COND - REAL

On exit, an estimate of the condition number of the factored matrix. Must be called after the numerical factorization subroutine, CGSSFA().

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-700 : Invalid calling sequence - need to call CGSSFA first.
-710 : Condition number estimate not available (not implemented
       for this HANDLE's matix type).
```

NAME

cgssda - Deallocate working storage for the general sparse solver.

SYNOPSIS

```
SUBROUTINE CGSSDA ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSDA - Deallocate dynamically allocated working storage.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
none
```

NAME

cgssfa - General sparse solver numeric factorization.

SYNOPSIS

```
SUBROUTINE CGSSFA ( NEQNS, COLSTR, ROWIND, VALUES, HANDLE, IER )

INTEGER           NEQNS, COLSTR(*), ROWIND(*), IER
COMPLEX           VALUES(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSFA - Numeric factorization of a sparse matrix.

PARAMETERS

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

`COLSTR(*)` - INTEGER array

On entry, `COLSTR(*)` is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

`ROWIND(*)` - INTEGER array

On entry, `ROWIND(*)` is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

`VALUES(*)` - COMPLEX array

On entry, `VALUES(*)` is an array of size COLSTR(NEQNS+1)-1, containing the numeric values of the sparse matrix to be factored. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-300 : Invalid calling sequence - need to call CGSSOR first.
-301 : Failure to dynamically allocate memory.
-666 : Internal error.
```

NAME

cgssfs - General sparse solver one call interface.

SYNOPSIS

```
SUBROUTINE SGSSFS ( MTXTYP, PIVOT , NEQNS, COLSTR, ROWIND,
                    VALUES, NRHS  , RHS   , LDRHS , ORDMTHD,
                    OUTUNT, MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), NRHS, LDRHS,
                  OUTUNT, MSGLVL, IER
CHARACTER*3       ORDMTHD
COMPLEX           VALUES(*), RHS(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSFS - General sparse solver one call interface.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, pivot specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

VALUES(*) - COMPLEX array

On entry, VALUES(*) is an array of size COLSTR(NEQNS+1)-1, containing the non-zero numeric values of the sparse matrix to be factored. Unchanged on exit.

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

RHS(*) - COMPLEX array

On entry, RHS(LDRHS,NRHS) contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see CGSSUO)
```

Unchanged on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array of containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros is less than NEQNS.
-201 : Failure to dynamically allocate memory.
-301 : Failure to dynamically allocate memory.
-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS
-666 : Internal error.

NAME

cgssin - Initialize the general sparse solver.

SYNOPSIS

```
SUBROUTINE CGSSIN ( MTXTYP, PIVOT, NEQNS, COLSTR, ROWIND, OUTUNT,
                    MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), OUTUNT, MSGLVL, IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSIN - Initialize the sparse solver and input the matrix structure.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, PIVOT specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros less than NEQNS.
```

NAME

cgssor - General sparse solver ordering and symbolic factorization.

SYNOPSIS

```
SUBROUTINE CGSSOR ( ORDMTHD, HANDLE, IER )

CHARACTER*3       ORDMTHD
INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSOR - Orders and symbolically factors a sparse matrix.

PARAMETERS

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see CGSSUO)
```

Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-200 : Invalid calling sequence - need to call CGSSIN first.
-201 : Failure to dynamically allocate memory.
-666 : Internal error.

NAME

cgssps - Print general sparse solver statics.

SYNOPSIS

```
SUBROUTINE CGSSPS ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSPS - Print solver statistics.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-800 : Invalid calling sequence - need to call CGSSSL first.
-899 : Printed solver statistics not supported this release.
```

NAME

cgssrp - Return permutation used by the general sparse solver.

SYNOPSIS

```
SUBROUTINE CGSSRP ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSRP - Returns the permutation used by the solver for the fill-reducing ordering.

PARAMETERS

PERM(NEQNS) - INTEGER array

Undefined on entry. PERM(NEQNS) is the permutation array used by the sparse solver for the fill-reducing ordering. Modified on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-600 : Invalid calling sequence - need to call CGSSOR first.
```

NAME

cgsssl - Solve routine for the general sparse solver.

SYNOPSIS

```
SUBROUTINE CGSSSL ( NRHS, RHS, LDRHS, HANDLE, IER )

INTEGER           NRHS, LDRHS, IER
COMPLEX           RHS(LDRHS,NRHS)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSSL - Triangular solve of a factored sparse matrix.

PARAMETERS

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

`RHS(LDRHS,*)` - COMPLEX array

On entry, `RHS(LDRHS,NRHS)` contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-400 : Invalid calling sequence - need to call CGSSFA first.
```

-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS

NAME

cgssuo - User supplied permutation for ordering used in the general sparse solver.

SYNOPSIS

```
SUBROUTINE CGSSUO ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

CGSSUO - User supplied permutation for ordering. Must be called after `CGSSIN()` (sparse solver initialization) and before `CGSSOR()` (sparse solver ordering).

PARAMETERS

`PERM(NEQNS)` - INTEGER array

On entry, `PERM(NEQNS)` is a permutation array supplied by the user for the fill-reducing ordering. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-500 : Invalid calling sequence - need to call CGSSIN first.
```

NAME

cgtcon - estimate the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF

SYNOPSIS

```
 SUBROUTINE CGTCON( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, INFO)
 CHARACTER * 1 NORM
 COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
 INTEGER N, INFO
 INTEGER IPIVOT(*)
 REAL ANORM, RCOND

 SUBROUTINE CGTCON_64( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, INFO)
 CHARACTER * 1 NORM
 COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*)
 REAL ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE GTCON( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: ANORM, RCOND

 SUBROUTINE GTCON_64( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

```
REAL :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void cgtcon(char norm, int n, complex *low, complex *diag, complex *up1, complex *up2, int *ipivot, float anorm, float *rcond, int *info);

void cgtcon_64(char norm, long n, complex *low, complex *diag, complex *up1, complex *up2, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

cgtcon estimates the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by CGTTRF.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.

- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

cgthr - Gathers specified elements from y into x.

SYNOPSIS

```
SUBROUTINE CGTHR(NZ, Y, X, INDX)

COMPLEX Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE CGTHR_64(NZ, Y, X, INDX)

COMPLEX Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHR([NZ], Y, X, INDX)

```
COMPLEX, DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHR_64([NZ], Y, X, INDX)

COMPLEX, DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CGTHR - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input)

Vector in full storage form. Unchanged on exit.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

cgthrz - Gather and zero.

SYNOPSIS

```
SUBROUTINE CGTHRZ(NZ, Y, X, INDX)

COMPLEX Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE CGTHRZ_64(NZ, Y, X, INDX)

COMPLEX Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHRZ([NZ], Y, X, INDX)

```
COMPLEX, DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHRZ_64([NZ], Y, X, INDX)

COMPLEX, DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CGTHRZ - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. The gathered elements of y are set to zero. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
  y(indx(i)) = 0
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input/output)

Vector in full storage form. Gathered elements are set to zero.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

cgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CGTRFS( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGTRFS_64( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GTRFS( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF, DIAGF,
*       UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GTRFS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgtrfs(char transa, int n, int nrhs, complex *low, complex *diag, complex *up, complex *lowf, complex *diagf, complex *upf1, complex *upf2, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cgtrfs_64(char transa, long n, long nrhs, complex *low, complex *diag, complex *up, complex *lowf, complex *diagf, complex *upf1, complex *upf2, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cgtrfs improves the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The diagonal elements of A.
- **UP (input)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by CGTTRF.
- **DIAGF (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UPF2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CGTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cgtsv - solve the equation A*X = B,

SYNOPSIS

```
SUBROUTINE CGTSV( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
COMPLEX LOW(*), DIAG(*), UP(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE CGTSV_64( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
COMPLEX LOW(*), DIAG(*), UP(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE GTSV( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
COMPLEX, DIMENSION(:) :: LOW, DIAG, UP
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE GTSV_64( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
COMPLEX, DIMENSION(:) :: LOW, DIAG, UP
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cgtsv(int n, int nrhs, complex *low, complex *diag, complex *up, complex *b, int ldb, int *info);

void cgtsv_64(long n, long nrhs, complex *low, complex *diag, complex *up, complex *b, long ldb, long *info);

PURPOSE

cgtsv solves the equation

where A is an N-by-N tridiagonal matrix, by Gaussian elimination with partial pivoting.

Note that the equation A'*X = B may be solved by interchanging the order of the arguments DU and DL.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) subdiagonal elements of A. On exit, LOW is overwritten by the (n-2) elements of the second superdiagonal of the upper triangular matrix U from the LU factorization of A, in LOW(1), ..., LOW(n-2).
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A. On exit, DIAG is overwritten by the n diagonal elements of U.
- **UP (input/output)**
  On entry, UP must contain the (n-1) superdiagonal elements of A. On exit, UP is overwritten by the (n-1) elements of the first superdiagonal of U.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero, and the solution
has not been computed.  The factorization has not been
completed unless i  = N.
```

NAME

cgtsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE CGTSVX( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CGTSVX_64( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GTSVX( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GTSVX_64( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP,
*       LOWF, DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cgtsvx(char fact, char transa, int n, int nrhs, complex *low, complex *diag, complex *up, complex *lowf, complex *diagf, complex *upf1, complex *upf2, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cgtsvx_64(char fact, char transa, long n, long nrhs, complex *low, complex *diag, complex *up, complex *lowf, complex *diagf, complex *upf1, complex *upf2, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cgtsvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a tridiagonal matrix of order N and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the LU decomposition is used to factor the matrix A as A = L * U, where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

2. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': LOWF, DIAGF, UPF1, UPF2, and IPIVOT contain the factored form of A; LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2 and IPIVOT will not be modified. = 'N': The matrix will be copied to LOWF, DIAGF, and UPF1 and factored.
- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The n diagonal elements of A.
- **UP (input)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input/output)**
  If FACT = 'F', then LOWF is an input argument and on entry contains the (n-1) multipliers that define the matrix L from the LU

factorization of A as computed by CGTTRF.

If FACT = 'N', then LOWF is an output argument and on exit contains the (n-1) multipliers that define the matrix L from the LU factorization of A.

- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

  If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input/output)**
  If FACT = 'F', then UPF1 is an input argument and on entry contains the (n-1) elements of the first superdiagonal of U.

  If FACT = 'N', then UPF1 is an output argument and on exit contains the (n-1) elements of the first superdiagonal of U.
- **UPF2 (input/output)**
  If FACT = 'F', then UPF2 is an input argument and on entry contains the (n-2) elements of the second superdiagonal of U.

  If FACT = 'N', then UPF2 is an output argument and on exit contains the (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the LU factorization of A as computed by CGTTRF.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the LU factorization of A; row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has not been completed unless i  = N, but the
factor U is exactly singular, so the solution
and error bounds could not be computed.
RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
```

to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of RCOND would suggest.

NAME

cgttrf - compute an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges

---

SYNOPSIS

```
SUBROUTINE CGTTRF( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE CGTTRF_64( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GTTRF( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GTTRF_64( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgttrf(int n, complex *low, complex *diag, complex *up1, complex *up2, int *ipivot, int *info);

void cgttrf_64(long n, complex *low, complex *diag, complex *up1, complex *up2, long *ipivot, long *info);

PURPOSE

cgttrf computes an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

```
A = L * U
```

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

ARGUMENTS

- **N (input)**
  The order of the matrix A.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input/output)**
  On entry, UP1 must contain the (n-1) super-diagonal elements of A.

  On exit, UP1 is overwritten by the (n-1) elements of the first super-diagonal of U.
- **UP2 (output)**
  On exit, UP2 is overwritten by the (n-2) elements of the second super-diagonal of U.
- **IPIVOT (output)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -k, the k-th argument had an illegal value

 > 0:  if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

cgttrs - solve one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE CGTTRS( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
 INTEGER N, NRHS, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE CGTTRS_64( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
 INTEGER*8 N, NRHS, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GTTRS( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2
 COMPLEX, DIMENSION(:,:) :: B
 INTEGER :: N, NRHS, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GTTRS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX, DIMENSION(:) :: LOW, DIAG, UP1, UP2
 COMPLEX, DIMENSION(:,:) :: B
 INTEGER(8) :: N, NRHS, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cgttrs(char transa, int n, int nrhs, complex *low, complex *diag, complex *up1, complex *up2, int *ipivot, complex *b, int ldb, int *info);

void cgttrs_64(char transa, long n, long nrhs, complex *low, complex *diag, complex *up1, complex *up2, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cgttrs solves one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B, with a tridiagonal matrix A using the LU factorization computed by CGTTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first super-diagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second super-diagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input/output)**
  On entry, the matrix of right hand side vectors B. On exit, B is overwritten by the solution vectors X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

chbev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE CHBEV( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, NDIAG, LDA, LDZ, INFO
 REAL W(*), WORK2(*)

 SUBROUTINE CHBEV_64( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, LDZ, INFO
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEV( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER :: N, NDIAG, LDA, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK2

 SUBROUTINE HBEV_64( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, NDIAG, LDA, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chbev(char jobz, char uplo, int n, int ndiag, complex *a, int lda, float *w, complex *z, int ldz, int *info);

void chbev_64(char jobz, char uplo, long n, long ndiag, complex *a, long lda, float *w, complex *z, long ldz, long *info);

PURPOSE

chbev computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG + 1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

chbevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE CHBEVD( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), Z(LDZ,*), WORK(*)
 INTEGER N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHBEVD_64( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEVD( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Z
 INTEGER :: N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HBEVD_64( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Z
 INTEGER(8) :: N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chbevd(char jobz, char uplo, int n, int kd, complex *ab, int ldab, float *w, complex *z, int ldz, complex *work, int lwork, float *rwork, int lrwork, int *info);

void chbevd_64(char jobz, char uplo, long n, long kd, complex *ab, long ldab, float *w, complex *z, long ldz, complex *work, long lwork, float *rwork, long lrwork, long *info);

PURPOSE

chbevd computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, AB is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows KD and KD+1 of AB, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of AB.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD + 1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N .

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

chbevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE CHBEVX( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ, VL,
*      VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)

 SUBROUTINE CHBEVX_64( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ,
*      VL, VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEVX( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q, [LDQ],
*       VL, VU, IL, IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [WORK2],
*       [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Q, Z
 INTEGER :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK2
```

```
 SUBROUTINE HBEVX_64( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q,
*       [LDQ], VL, VU, IL, IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK],
*       [WORK2], [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Q, Z
 INTEGER(8) :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chbevx(char jobz, char range, char uplo, int n, int ndiag, complex *a, int lda, complex *q, int ldq, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, complex *z, int ldz, int *ifail, int *info);

void chbevx_64(char jobz, char range, char uplo, long n, long ndiag, complex *a, long lda, complex *q, long ldq, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

chbevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
  will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG + 1.
- **Q (output)**
  If JOBZ = 'V', the N-by-N unitary matrix used in the reduction to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'V', then LDQ > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

chbgst - reduce a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

---

SYNOPSIS

```
 SUBROUTINE CHBGST( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*      LDX, WORK, RWORK, INFO)
 CHARACTER * 1 VECT, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL RWORK(*)

 SUBROUTINE CHBGST_64( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*      LDX, WORK, RWORK, INFO)
 CHARACTER * 1 VECT, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGST( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       X, [LDX], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, X
 INTEGER :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE HBGST_64( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       X, [LDX], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, X
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void chbgst(char vect, char uplo, int n, int ka, int kb, complex *ab, int ldab, complex *bb, int ldbb, complex *x, int ldx, int *info);

void chbgst_64(char vect, char uplo, long n, long ka, long kb, complex *ab, long ldab, complex *bb, long ldbb, complex *x, long ldx, long *info);

PURPOSE

chbgst reduces a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y, such that C has the same bandwidth as A.

B must have been previously factorized as S**H*S by CPBSTF, using a split Cholesky factorization. A is overwritten by C = X**H*A*X, where X = S**(-1)*Q and Q is a unitary matrix chosen to preserve the bandwidth of A.

ARGUMENTS

- **VECT (input)**

  = 'N': do not form the transformation matrix X;

  = 'V': form X.
- **UPLO (input)**

  = 'U': Upper triangle of A is stored;

  = 'L': Lower triangle of A is stored.
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = KB > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the transformed matrix X**H*A*X, stored in the same format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input)**
  The banded factor S from the split Cholesky factorization of B, as returned by CPBSTF, stored in the first kb+1 rows of the array.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **X (output)**
  If VECT = 'V', the n-by-n matrix X. If VECT = 'N', the array X is not referenced.
- **LDX (input)**
  The leading dimension of the array X. LDX > = `max(1,N)` if VECT = 'V'; LDX > = 1 otherwise.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

chbgv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE CHBGV( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*       LDZ, WORK, RWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL W(*), RWORK(*)

 SUBROUTINE CHBGV_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*       Z, LDZ, WORK, RWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGV( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB], W,
*        Z, [LDZ], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGV_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        W, Z, [LDZ], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chbgv(char jobz, char uplo, int n, int ka, int kb, complex *ab, int ldab, complex *bb, int ldbb, float *w, complex *z, int ldz, int *info);

void chbgv_64(char jobz, char uplo, long n, long ka, long kb, complex *ab, long ldab, complex *bb, long ldbb, float *w, complex *z, long ldz, long *info);

PURPOSE

chbgv computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The

j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` < =i < =j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j < =i < =min(n,j+kb).

On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

NAME

chbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE CHBGVD( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*      LDZ, WORK, LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHBGVD_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*      Z, LDZ, WORK, LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGVD( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGVD_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK],
```

```
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chbgvd(char jobz, char uplo, int n, int ka, int kb, complex *ab, int ldab, complex *bb, int ldbb, float *w, complex *z, int ldz, int *info);

void chbgvd_64(char jobz, char uplo, long n, long ka, long kb, complex *ab, long ldab, complex *bb, long ldbb, float *w, complex *z, long ldz, long *info);

PURPOSE

chbgvd computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > =

0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.

- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= N. If JOBZ = 'V' and N > 1, LWORK >= 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **RWORK (workspace)**
  On exit, if INFO =0, `RWORK(1)` returns the optimal LRWORK.

- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK >= 1. If JOBZ = 'N' and N > 1, LRWORK >= N. If JOBZ = 'V' and N > 1, LRWORK >= 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO =0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

chbgvx - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE CHBGVX( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB, LDBB,
*      Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)

 SUBROUTINE CHBGVX_64( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB,
*      LDBB, Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK,
*      IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGVX( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Q, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
```

```
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGVX_64( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, BB, Q, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chbgvx(char jobz, char range, char uplo, int n, int ka, int kb, complex *ab, int ldab, complex *bb, int ldbb, complex *q, int ldq, float vl, float vu, int il, int iu, float abstol, int *m, float *w, complex *z, int ldz, int *ifail, int *info);

void chbgvx_64(char jobz, char range, char uplo, long n, long ka, long kb, complex *ab, long ldab, complex *bb, long ldbb, complex *q, long ldq, float vl, float vu, long il, long iu, float abstol, long *m, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

chbgvx computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
 will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangles of A and B are stored;
```

= 'L': Lower triangles of A and B are stored.

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **Q (output)**
  If JOBZ = 'V', the n-by-n matrix used in the reduction of A*x = (lambda)*B*x to standard form, i.e. C*x = (lambda)*x, and consequently C to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'N', LDQ >= 1. If JOBZ = 'V', LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing AP to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  then i eigenvectors failed to converge.  Their
indices are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

  returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

chbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE CHBMV( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER N, NDIAG, LDA, INCX, INCY

 SUBROUTINE CHBMV_64( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA,
*      Y, INCY)
 CHARACTER * 1 UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 N, NDIAG, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE HBMV( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX], BETA,
*       Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INCX, INCY

 SUBROUTINE HBMV_64( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:) :: X, Y
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, NDIAG, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void chbmv(char uplo, int n, int ndiag, complex alpha, complex *a, int lda, complex *x, int incx, complex beta, complex *y, int incy);

void chbmv_64(char uplo, long n, long ndiag, complex alpha, complex *a, long lda, complex *x, long incx, complex beta, complex *y, long incy);

PURPOSE

chbmv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with ndiag super-diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the band matrix A is being supplied as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is being supplied.

  UPLO = 'L' or 'l' The lower triangular part of A is being supplied.

  Unchanged on exit.

- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.

- **NDIAG (input)**
  On entry, NDIAG specifies the number of super-diagonals of the matrix A. NDIAG must satisfy 0 .le. NDIAG. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.

- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the upper triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

  Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag

triangle of the array A is not referenced. The following program segment will transfer the lower triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

chbtrd - reduce a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
 SUBROUTINE CHBTRD( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ, WORK,
*      INFO)
 CHARACTER * 1 VECT, UPLO
 COMPLEX AB(LDAB,*), Q(LDQ,*), WORK(*)
 INTEGER N, KD, LDAB, LDQ, INFO
 REAL D(*), E(*)

 SUBROUTINE CHBTRD_64( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ,
*      WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 COMPLEX AB(LDAB,*), Q(LDQ,*), WORK(*)
 INTEGER*8 N, KD, LDAB, LDQ, INFO
 REAL D(*), E(*)
```

F95 INTERFACE

```
 SUBROUTINE HBTRD( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Q
 INTEGER :: N, KD, LDAB, LDQ, INFO
 REAL, DIMENSION(:) :: D, E

 SUBROUTINE HBTRD_64( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: AB, Q
 INTEGER(8) :: N, KD, LDAB, LDQ, INFO
```

```
REAL, DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void chbtrd(char vect, char uplo, int n, int kd, complex *ab, int ldab, float *d, float *e, complex *q, int ldq, int *info);

void chbtrd_64(char vect, char uplo, long n, long kd, complex *ab, long ldab, float *d, float *e, complex *q, long ldq, long *info);

PURPOSE

chbtrd reduces a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **VECT (input)**

  ```
  = 'N':  do not form Q;

  = 'V':  form Q;

  = 'U':  update a matrix X, by forming X*Q.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd). On exit, the diagonal elements of AB are overwritten by the diagonal elements of the tridiagonal matrix T; if KD > 0, the elements on the first superdiagonal (if UPLO = 'U') or the first subdiagonal (if UPLO = 'L') are overwritten by the off-diagonal elements of T; the rest of AB is overwritten by values generated during the reduction.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD+1.

- **D (output)**
  The diagonal elements of the tridiagonal matrix T.

- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `T(i,i+1)` if UPLO = 'U'; `E(i)` = `T(i+1,i)` if UPLO = 'L'.
- **Q (input/output)**
  On entry, if VECT = 'U', then Q must contain an N-by-N matrix X; if VECT = 'N' or 'V', then Q need not be set.

  On exit: if VECT = 'V', Q contains the N-by-N unitary matrix Q; if VECT = 'U', Q contains the product X*Q; if VECT = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1, and LDQ >= N if VECT = 'V' or 'U'.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Modified by Linda Kaufman, Bell Labs.

NAME

checon - estimate the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
 SUBROUTINE CHECON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*)
 REAL ANORM, RCOND

 SUBROUTINE CHECON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*)
 REAL ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE HECON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: ANORM, RCOND

 SUBROUTINE HECON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void checon(char uplo, int n, complex *a, int lda, int *ipivot, float anorm, float *rcond, int *info);

void checon_64(char uplo, long n, complex *a, long lda, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

checon estimates the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cheev - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE CHEEV( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, WORK2,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, LDWORK, INFO
 REAL W(*), WORK2(*)

 SUBROUTINE CHEEV_64( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, WORK2,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, LDWORK, INFO
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEV( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK2

 SUBROUTINE HEEV_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void cheev(char jobz, char uplo, int n, complex *a, int lda, float *w, int *info);

void cheev_64(char jobz, char uplo, long n, complex *a, long lda, float *w, long *info);

PURPOSE

cheev computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,2*N-1). For optimal efficiency, LDWORK >= (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **WORK2 (workspace)**

dimension(max(1,3*N-2))

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

cheevd - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE CHEEVD( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHEEVD_64( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVD( JOBZ, UPLO, [N], A, [LDA], W, WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HEEVD_64( JOBZ, UPLO, [N], A, [LDA], W, WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
```

```
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void cheevd(char jobz, char uplo, int n, complex *a, int lda, float *w, complex *work, int lwork, float *rwork, int lrwork, int *info);

void cheevd_64(char jobz, char uplo, long n, complex *a, long lda, float *w, complex *work, long lwork, float *rwork, long lrwork, long *info);

PURPOSE

cheevd computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N + 1. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, RWORK(1) returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If N < = 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK must be at least 1. If JOBZ = 'N' and N > 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Jeff Rutter, Computer Science Division, University of California
at Berkeley, USA
```

NAME

cheevr - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE CHEEVR( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, RWORK, LRWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)

 SUBROUTINE CHEEVR_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, RWORK, LRWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVR( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [RWORK], [LRWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

```
 SUBROUTINE HEEVR_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [RWORK],
*       [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void cheevr(char jobz, char range, char uplo, int n, complex *a, int lda, float vl, float vu, int il, int iu, float abstol, int *m, float *w, complex *z, int ldz, int *isuppz, int *info);

void cheevr_64(char jobz, char range, char uplo, long n, complex *a, long lda, float vl, float vu, long il, long iu, float abstol, long *m, float *w, complex *z, long ldz, long *isuppz, long *info);

PURPOSE

cheevr computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, CHEEVR calls CSTEGR to compute the

eigenspectrum using Relatively Robust Representations. CSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
   (a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
        is a relatively robust representation,

   (b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
       relative accuracy by the dqds algorithm,

   (c) If there is a cluster of close eigenvalues, "choose" sigma_i
       close to the cluster, and go to step (a),

   (d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
       compute the corresponding eigenvector by forming a
       rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : CHEEVR calls CSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. CHEEVR calls SSTEBZ and CSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of CSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is

determined to lie in an interval [a,b] of width less than or equal to

ABSTOL + EPS * max(|a|,|b|) ,

where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but furutre releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,2*N). For optimal efficiency, LWORK > = (NB+1)*N, where NB is the max of the blocksize for CHETRD and for CUNMTR as returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal (and minimal) LRWORK.
- **LRWORK (input)**
  The length of the array RWORK. LRWORK > = max(1,24*N).

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal (and minimal) LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK > = max(1,10*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: Internal error

FURTHER DETAILS

Based on contributions by

Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA

NAME

cheevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE CHEEVX( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)

 SUBROUTINE CHEEVX_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVX( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [LDWORK], [WORK2], [IWORK3],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK2
```

```
 SUBROUTINE HEEVX_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [LDWORK], [WORK2],
*       [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void cheevx(char jobz, char range, char uplo, int n, complex *a, int lda, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, complex *z, int ldz, int *ifail, int *info);

void cheevx_64(char jobz, char range, char uplo, long n, complex *a, long lda, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

cheevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  On normal exit, the first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK > = max(1,2*N). For optimal efficiency, LDWORK > = (NB+1)*N, where NB is the max of the blocksize for CHETRD and for CUNMTR as returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued

by XERBLA.

- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

chegs2 - reduce a complex Hermitian-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE CHEGS2( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER ITYPE, N, LDA, LDB, INFO

SUBROUTINE CHEGS2_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 ITYPE, N, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE HEGS2( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: ITYPE, N, LDA, LDB, INFO

SUBROUTINE HEGS2_64( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void chegs2(int itype, char uplo, int n, complex *a, int lda, complex *b, int ldb, int *info);

void chegs2_64(long itype, char uplo, long n, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

chegs2 reduces a complex Hermitian-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U')*A*inv(U)` or `inv(L)*A*inv(L')`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U` or L'*A*L.

B must have been previously factorized as U'*U or L*L' by CPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U')*A*inv(U) or inv(L)*A*inv(L');

  = 2 or 3: compute U*A*U' or L'*A*L.
  ```

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored, and how B has been factorized. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by CPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

chegst - reduce a complex Hermitian-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE CHEGST( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER ITYPE, N, LDA, LDB, INFO

SUBROUTINE CHEGST_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 ITYPE, N, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE HEGST( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: ITYPE, N, LDA, LDB, INFO

SUBROUTINE HEGST_64( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void chegst(int itype, char uplo, int n, complex *a, int lda, complex *b, int ldb, int *info);

void chegst_64(long itype, char uplo, long n, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

chegst reduces a complex Hermitian-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**H)*A*inv(U)` or `inv(L)*A*inv(L**H)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**H or L**H*A*L.

B must have been previously factorized as U**H*U or L*L**H by CPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U**H)*A*inv(U) or inv(L)*A*inv(L**H);

  = 2 or 3: compute U*A*U**H or L**H*A*L.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored and B is factored as
U**H*U;
  = 'L':  Lower triangle of A is stored and B is factored as
L*L**H.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by CPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

chegv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHEGV( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL W(*), WORK2(*)

 SUBROUTINE CHEGV_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGV( ITYPE, JOBZ, UPLO, N, A, [LDA], B, [LDB], W, [WORK],
*       [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK2

 SUBROUTINE HEGV_64( ITYPE, JOBZ, UPLO, N, A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chegv(int itype, char jobz, char uplo, int n, complex *a, int lda, complex *b, int ldb, float *w, int *info);

void chegv_64(long itype, char jobz, char uplo, long n, complex *a, long lda, complex *b, long ldb, float *w, long *info);

PURPOSE

chegv computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also

positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the Hermitian positive definite matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO <= N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,2*N-1). For optimal efficiency, LDWORK >= (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPOTRF or CHEEV returned an error code:

 < = N:  if INFO  = i, CHEEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

chegvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHEGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHEGVD_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGVD( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HEGVD_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
```

```
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chegvd(int itype, char jobz, char uplo, int n, complex *a, int lda, complex *b, int ldb, float *w, int *info);

void chegvd_64(long itype, char jobz, char uplo, long n, complex *a, long lda, complex *b, long ldb, float *w, long *info);

PURPOSE

chegvd computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the Hermitian matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. If N < = 1, LWORK > = 1. If JOBZ = 'N' and N > 1, LWORK > = N + 1. If JOBZ = 'V' and N > 1, LWORK > = 2*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If N < = 1, LRWORK > = 1. If JOBZ = 'N' and N > 1, LRWORK > = N. If JOBZ = 'V' and N > 1, LRWORK > = 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK > = 1. If JOBZ = 'N' and N > 1, LIWORK > = 1. If JOBZ = 'V' and N > 1, LIWORK > = 3 + 5*N.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPOTRF or CHEEVD returned an error code:

 < = N:  if INFO  = i, CHEEVD failed to converge;
i off-diagonal elements of an intermediate
```

```
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

chegvx - compute selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHEGVX( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB, VL,
*      VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), B(LDB,*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)

 SUBROUTINE CHEGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB,
*      VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(LDA,*), B(LDB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGVX( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B, [LDB],
*       VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, Z
 INTEGER :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
```

```
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HEGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B,
*       [LDB], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, Z
 INTEGER(8) :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chegvx(int itype, char jobz, char range, char uplo, int n, complex *a, int lda, complex *b, int ldb, float vl, float vu, int il, int iu, float abstol, int *m, float *w, complex *z, int ldz, int *ifail, int *info);

void chegvx_64(long itype, char jobz, char range, char uplo, long n, complex *a, long lda, complex *b, long ldb, float vl, float vu, long il, long iu, float abstol, long *m, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

chegvx computes selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the Hermitian matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**

The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.

- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

  If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,2*N-1). For optimal efficiency, LWORK > = (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPOTRF or CHEEVX returned an error code:

 < = N:  if INFO  = i, CHEEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

chemm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE CHEMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, LDA, LDB, LDC

 SUBROUTINE CHEMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE HEMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, LDA, LDB, LDC

 SUBROUTINE HEMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void chemm(char side, char uplo, int m, int n, complex alpha, complex *a, int lda, complex *b, int ldb, complex beta, complex *c, int ldc);

void chemm_64(char side, char uplo, long m, long n, complex alpha, complex *a, long lda, complex *b, long ldb, complex beta, complex *c, long ldc);

PURPOSE

chemm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the hermitian matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the hermitian matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the hermitian matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the hermitian matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the hermitian matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the hermitian matrix, such that when

UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced.

Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB must be at least max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry.

  On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, m). Unchanged on exit.

NAME

chemv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE CHEMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
COMPLEX ALPHA, BETA
COMPLEX A(LDA,*), X(*), Y(*)
INTEGER N, LDA, INCX, INCY

SUBROUTINE CHEMV_64( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
COMPLEX ALPHA, BETA
COMPLEX A(LDA,*), X(*), Y(*)
INTEGER*8 N, LDA, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE HEMV( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCX, INCY

SUBROUTINE HEMV_64( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void chemv(char uplo, int n, complex alpha, complex *a, int lda, complex *x, int incx, complex beta, complex *y, int incy);

void chemv_64(char uplo, long n, complex alpha, complex *a, long lda, complex *x, long incx, complex beta, complex *y, long incy);

PURPOSE

chemv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

cher - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

SYNOPSIS

```
SUBROUTINE CHER( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
COMPLEX X(*), A(LDA,*)
INTEGER N, INCX, LDA
REAL ALPHA

SUBROUTINE CHER_64( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
COMPLEX X(*), A(LDA,*)
INTEGER*8 N, INCX, LDA
REAL ALPHA
```

F95 INTERFACE

```
SUBROUTINE HER( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: X
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, INCX, LDA
REAL :: ALPHA

SUBROUTINE HER_64( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: X
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, INCX, LDA
REAL :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void cher(char uplo, int n, float alpha, complex *x, int incx, complex *a, int lda);

void cher_64(char uplo, long n, float alpha, complex *x, long incx, complex *a, long lda);

PURPOSE

cher performs the hermitian rank 1 operation A := alpha*x*conjg(x') + A where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

cher2 - perform the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

SYNOPSIS

```
SUBROUTINE CHER2( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER N, INCX, INCY, LDA

SUBROUTINE CHER2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
COMPLEX ALPHA
COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE HER2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, INCX, INCY, LDA

SUBROUTINE HER2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void cher2(char uplo, int n, complex alpha, complex *x, int incx, complex *y, int incy, complex *a, int lda);

void cher2_64(char uplo, long n, complex alpha, complex *x, long incx, complex *y, long incy, complex *a, long lda);

PURPOSE

cher2 performs the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

cher2k - perform one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C

SYNOPSIS

```
 SUBROUTINE CHER2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER N, K, LDA, LDB, LDC
 REAL BETA

 SUBROUTINE CHER2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDB, LDC
 REAL BETA
```

F95 INTERFACE

```
 SUBROUTINE HER2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: N, K, LDA, LDB, LDC
 REAL :: BETA

 SUBROUTINE HER2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: N, K, LDA, LDB, LDC
```

```
REAL :: BETA
```

C INTERFACE

#include <sunperf.h>

void cher2k(char uplo, char transa, int n, int k, complex alpha, complex *a, int lda, complex *b, int ldb, float beta, complex *c, int ldc);

void cher2k_64(char uplo, char transa, long n, long k, complex alpha, complex *a, long lda, complex *b, long ldb, float beta, complex *c, long ldc);

PURPOSE

cher2k K performs one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C where alpha and beta are scalars with beta real, C is an n by n Hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'C' or 'c', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**

On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

  Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

cherfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CHERFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CHERFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HERFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HERFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cherfs(char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cherfs_64(char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cherfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CHETRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cherk - perform one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C

SYNOPSIS

```
 SUBROUTINE CHERK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX A(LDA,*), C(LDC,*)
 INTEGER N, K, LDA, LDC
 REAL ALPHA, BETA

 SUBROUTINE CHERK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX A(LDA,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDC
 REAL ALPHA, BETA
```

F95 INTERFACE

```
 SUBROUTINE HERK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: N, K, LDA, LDC
 REAL :: ALPHA, BETA

 SUBROUTINE HERK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: N, K, LDA, LDC
 REAL :: ALPHA, BETA
```

C INTERFACE

#include <sunperf.h>

void cherk(char uplo, char transa, int n, int k, float alpha, complex *a, int lda, float beta, complex *c, int ldc);

void cherk_64(char uplo, char transa, long n, long k, float alpha, complex *a, long lda, float beta, complex *c, long ldc);

PURPOSE

cherk performs one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C where alpha and beta are real scalars, C is an n by n Hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*conjg(A') + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*conjg(A')*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'C' or 'c', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

  Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

chesv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CHESV( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE CHESV_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE HESV( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE HESV_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chesv(char uplo, int n, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void chesv_64(char uplo, long n, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

chesv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**H,  if UPLO = 'U', or

A = L * D * L**H,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CHETRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and

D(k-1:k,k-1:k) is a 2-by-2 diagonal block. If UPLO = 'L' and IPIVOT(k) = IPIVOT(k+1) < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and D(k:k+1,k:k+1) is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 1, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CHETRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

chesvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CHESVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CHESVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HESVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HESVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
```

```
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void chesvx(char fact, char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void chesvx_64(char fact, char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

chesvx uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
A = U * D * U**H,  if UPLO = 'U', or

A = L * D * L**H,  if UPLO = 'L',

where U (or L) is a product of permutation and unit upper (lower)
triangular matrices, and D is Hermitian and block diagonal with
1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CHETRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CHETRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 2*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CHETRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

chetf2 - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CHETF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE CHETF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE HETF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE HETF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void chetf2(char uplo, int n, complex *a, int lda, int *ipiv, int *info);

void chetf2_64(char uplo, long n, complex *a, long lda, long *ipiv, long *info);

PURPOSE

chetf2 computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the conjugate transpose of U, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by

```
  J. Lewis, Boeing Computer Services Company

  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

If UPLO = 'U', then A = U*D*U', where

```
   U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
            (   I    v    0   )   k-s

   U(k)  =  (   0    I    0   )   s

            (   0    0    I   )   n-k

               k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
   L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
            (   I    0     0   )  k-1

   L(k)  =  (   0    I     0   )  s

            (   0    v     I   )  n-k-s+1

               k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

chetrd - reduce a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
SUBROUTINE CHETRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, LDA, LWORK, INFO
REAL D(*), E(*)

SUBROUTINE CHETRD_64( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, LDA, LWORK, INFO
REAL D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE HETRD( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: D, E

SUBROUTINE HETRD_64( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void chetrd(char uplo, int n, complex *a, int lda, float *d, float *e, complex *tau, int *info);

void chetrd_64(char uplo, long n, complex *a, long lda, float *d, float *e, complex *tau, long *info);

PURPOSE

chetrd reduces a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. For optimum performance LWORK > = N*NB, where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

chetrf - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CHETRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHETRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chetrf(char uplo, int n, complex *a, int lda, int *ipivot, int *info);

void chetrf_64(char uplo, long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

chetrf computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**H  or  A = L*D*L**H
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

chetri - compute the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
SUBROUTINE CHETRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHETRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chetri(char uplo, int n, complex *a, int lda, int *ipivot, int *info);

void chetri_64(char uplo, long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

chetri computes the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.

  On exit, if INFO = 0, the (Hermitian) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

chetrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
SUBROUTINE CHETRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHETRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chetrs(char uplo, int n, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void chetrs_64(char uplo, long n, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

chetrs solves a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

chgeqz - implement a single-shift version of the QZ method for finding the generalized eigenvalues `w(i)=ALPHA(i)/BETA(i)` of the equation det(A-w(i) B) = 0 If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form (i.e., A and B are both upper triangular) by applying one unitary tranformation (usually called Q) on the left and another (usually called Z) on the right

---

SYNOPSIS

```
 SUBROUTINE CHGEQZ( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL RWORK(*)

 SUBROUTINE CHGEQZ_64( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B,
*      LDB, ALPHA, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HGEQZ( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE HGEQZ_64( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [INFO])
```

```
CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void chgeqz(char job, char compq, char compz, int n, int ilo, int ihi, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *q, int ldq, complex *z, int ldz, int *info);

void chgeqz_64(char job, char compq, char compz, long n, long ilo, long ihi, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *q, long ldq, complex *z, long ldz, long *info);

PURPOSE

chgeqz implements a single-shift version of the QZ method for finding the generalized eigenvalues `w(i)=ALPHA(i)/BETA(i)` of the equation A are then ALPHA(1),...,ALPHA(N), and of B are BETA(1),...,BETA(N).

If JOB='S' and COMPQ and COMPZ are 'V' or 'I', then the unitary transformations used to reduce (A,B) are accumulated into the arrays Q and Z s.t.:

(in) `A(in)` Z(in)* = `Q(out)` `A(out)` Z(out)* (in) `B(in)` Z(in)* = `Q(out)` `B(out)` Z(out)*

Ref: C.B. Moler & G.W. Stewart, ``An Algorithm for Generalized Matrixigenvalue Problems", SIAM J. Numer. Anal., 10(1973),p. 241--256.

ARGUMENTS

- **JOB (input)**

```
 = 'E': compute only ALPHA and BETA.  A and B will not
necessarily be put into generalized Schur form.
 = 'S': put A and B into generalized Schur form, as well
as computing ALPHA and BETA.
```

- **COMPQ (input)**

```
 = 'N': do not modify Q.

 = 'V': multiply the array Q on the right by the conjugate
transpose of the unitary tranformation that is
applied to the left side of A and B to reduce them
to Schur form.
 = 'I': like COMPQ ='V', except that Q will be initialized to
the identity first.
```

- **COMPZ (input)**

```
 = 'N': do not modify Z.

 = 'V': multiply the array Z on the right by the unitary
tranformation that is applied to the right side of
A and B to reduce them to Schur form.
 = 'I': like COMPZ ='V', except that Z will be initialized to
the identity first.
```

- **N (input)**
  The order of the matrices A, B, Q, and Z. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **A (input/output)**
  On entry, the N-by-N upper Hessenberg matrix A. Elements below the subdiagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to upper triangular form. If JOB ='E', then on exit A will have been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. Elements below the diagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to upper triangular form. If JOB ='E', then on exit B will have been destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **ALPHA (output)**
  The diagonal elements of A when the pair (A,B) has been reduced to Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues.
- **BETA (output)**
  The diagonal elements of B when the pair (A,B) has been reduced to Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues. A and B are normalized so that `BETA(1),...,BETA(N)` are non-negative real numbers.
- **Q (input/output)**
  If COMPQ ='N', then Q will not be referenced. If COMPQ ='V' or 'I', then the conjugate transpose of the unitary transformations which are applied to A and B on the left will be applied to the array Q on the right.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If COMPQ ='V' or 'I', then LDQ > = N.
- **Z (input/output)**
  If COMPZ ='N', then Z will not be referenced. If COMPZ ='V' or 'I', then the unitary transformations which are applied to A and B on the right will be applied to the array Z on the right.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If COMPZ ='V' or 'I', then LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO > = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**

```
dimension(N)
```

- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1,...,N: the QZ iteration did not converge.  (A,B) is not
in Schur form, but ALPHA(i) and BETA(i),
i =INFO+1,...,N should be correct.
 = N+1,...,2*N: the shift calculation failed.  (A,B) is not
in Schur form, but ALPHA(i) and BETA(i),
i =INFO-N+1,...,N should be correct.
 > 2*N:     various "impossible" errors.
```

FURTHER DETAILS

We assume that complex ABS works as long as its value is less than overflow.

NAME

chpcon - estimate the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE CHPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)
REAL ANORM, RCOND

SUBROUTINE CHPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
REAL ANORM, RCOND
```

F95 INTERFACE

```
SUBROUTINE HPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND

SUBROUTINE HPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void chpcon(char uplo, int n, complex *a, int *ipivot, float anorm, float *rcond, int *info);

void chpcon_64(char uplo, long n, complex *a, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

chpcon estimates the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

chpev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage

---

SYNOPSIS

```
SUBROUTINE CHPEV( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, WORK2, INFO)
CHARACTER * 1 JOBZ, UPLO
COMPLEX A(*), Z(LDZ,*), WORK(*)
INTEGER N, LDZ, INFO
REAL W(*), WORK2(*)

SUBROUTINE CHPEV_64( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, WORK2, INFO)
CHARACTER * 1 JOBZ, UPLO
COMPLEX A(*), Z(LDZ,*), WORK(*)
INTEGER*8 N, LDZ, INFO
REAL W(*), WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE HPEV( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
COMPLEX, DIMENSION(:) :: A, WORK
COMPLEX, DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: W, WORK2

SUBROUTINE HPEV_64( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
COMPLEX, DIMENSION(:) :: A, WORK
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chpev(char jobz, char uplo, int n, complex *a, float *w, complex *z, int ldz, int *info);

void chpev_64(char jobz, char uplo, long n, complex *a, float *w, complex *z, long ldz, long *info);

PURPOSE

chpev computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  `dimension(MAX(1,2*N-1))`

- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`

- **INFO (output)**

= 0: successful exit.

< 0: if INFO = -i, the i-th argument had an illegal value.

> 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

chpevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE CHPEVD( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AP(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHPEVD_64( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK,
*      RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPEVD( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: AP, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HPEVD_64( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: AP, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER(8) :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chpevd(char jobz, char uplo, int n, complex *ap, float *w, complex *z, int ldz, complex *work, int lwork, float *rwork, int lrwork, int *info);

void chpevd_64(char jobz, char uplo, long n, complex *ap, float *w, complex *z, long ldz, complex *work, long lwork, float *rwork, long lrwork, long *info);

PURPOSE

chpevd computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, AP is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N < = 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N < = 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

chpevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE CHPEVX( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU, ABTOL,
*       NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(*), Z(LDZ,*), WORK(*)
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)

 SUBROUTINE CHPEVX_64( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX A(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPEVX( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU, ABTOL,
*       [NFOUND], W, Z, [LDZ], [WORK], [WORK2], [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: A, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK2

 SUBROUTINE HPEVX_64( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU,
*       ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [WORK2], [IWORK3], IFAIL,
*       [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
COMPLEX, DIMENSION(:) :: A, WORK
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
REAL :: VL, VU, ABTOL
REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chpevx(char jobz, char range, char uplo, int n, complex *a, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, complex *z, int ldz, int *ifail, int *info);

void chpevx_64(char jobz, char range, char uplo, long n, complex *a, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

chpevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues/vectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found;

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO

= 'L', A(i + (j-1)*(2*n-j)/2) = A(i,j) for j < =i < =n.

On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  If INFO = 0, the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.

- **INFO (output)**

```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

chpgst - reduce a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage

---

SYNOPSIS

```
SUBROUTINE CHPGST( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), BP(*)
INTEGER ITYPE, N, INFO

SUBROUTINE CHPGST_64( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), BP(*)
INTEGER*8 ITYPE, N, INFO
```

F95 INTERFACE

```
SUBROUTINE HPGST( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, BP
INTEGER :: ITYPE, N, INFO

SUBROUTINE HPGST_64( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, BP
INTEGER(8) :: ITYPE, N, INFO
```

C INTERFACE

#include <sunperf.h>

void chpgst(int itype, char uplo, int n, complex *ap, complex *bp, int *info);

void chpgst_64(long itype, char uplo, long n, complex *ap, complex *bp, long *info);

PURPOSE

chpgst reduces a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**H)*A*inv(U)` or `inv(L)*A*inv(L**H)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**H or L**H*A*L.

B must have been previously factorized as U**H*U or L*L**H by CPPTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U**H)*A*inv(U) or inv(L)*A*inv(L**H);

  = 2 or 3: compute U*A*U**H or L**H*A*L.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored and B is factored as
U**H*U;
  = 'L':  Lower triangle of A is stored and B is factored as
L*L**H.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **BP (input)**
  The triangular factor from the Cholesky factorization of B, stored in the same format as A, as returned by CPPTRF.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

chpgv - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHPGV( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(*), B(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDZ, INFO
 REAL W(*), WORK2(*)

 SUBROUTINE CHPGV_64( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX A(*), B(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDZ, INFO
 REAL W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGV( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: A, B, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK2

 SUBROUTINE HPGV_64( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: A, B, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER(8) :: ITYPE, N, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void chpgv(int itype, char jobz, char uplo, int n, complex *a, complex *b, float *w, complex *z, int ldz, int *info);

void chpgv_64(long itype, char jobz, char uplo, long n, complex *a, complex *b, float *w, complex *z, long ldz, long *info);

PURPOSE

chpgv computes all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of A are destroyed.

- **B (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array B as follows: if UPLO = 'U', B(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', B(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(MAX(1,2*N-1))`
- **WORK2 (workspace)**
  `dimension(MAX(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEV returned an error code:

 < = N:  if INFO  = i, CHPEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not convergeto zero;
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

chpgvd - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHPGVD( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL W(*), RWORK(*)

 SUBROUTINE CHPGVD_64( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGVD( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ], [WORK],
*       [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: AP, BP, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, RWORK

 SUBROUTINE HPGVD_64( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ],
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
```

```
COMPLEX, DIMENSION(:) :: AP, BP, WORK
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chpgvd(int itype, char jobz, char uplo, int n, complex *ap, complex *bp, float *w, complex *z, int ldz, int *info);

void chpgvd_64(long itype, char jobz, char uplo, long n, complex *ap, complex *bp, float *w, complex *z, long ldz, long *info);

PURPOSE

chpgvd computes all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;
  ```

= 'L': Lower triangles of A and B are stored.

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 <=i <=j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j <=i <=n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= N. If JOBZ = 'V' and N > 1, LWORK >= 2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK >= 1. If JOBZ = 'N' and N > 1, LRWORK >= N. If JOBZ = 'V' and N > 1, LRWORK >= 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

= 0: successful exit

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEVD returned an error code:

 < = N:  if INFO  = i, CHPEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not convergeto zero;
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

chpgvx - compute selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE CHPGVX( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU, IL,
*      IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)

 SUBROUTINE CHPGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU,
*      IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGVX( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], [WORK], [RWORK], [IWORK], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: AP, BP, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

```
 SUBROUTINE HPGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU,
*       IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [RWORK], [IWORK], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX, DIMENSION(:) :: AP, BP, WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER(8) :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void chpgvx(int itype, char jobz, char range, char uplo, int n, complex *ap, complex *bp, float vl, float vu, int il, int iu, float abstol, int *m, float *w, complex *z, int ldz, int *ifail, int *info);

void chpgvx_64(long itype, char jobz, char range, char uplo, long n, complex *ap, complex *bp, float vl, float vu, long il, long iu, float abstol, long *m, float *w, complex *z, long ldz, long *ifail, long *info);

PURPOSE

chpgvx computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;
  ```

```
 = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
 = 'U':  Upper triangles of A and B are stored;

 = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing AP to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z

holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I.

If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEVX returned an error code:

 < = N:  if INFO  = i, CHPEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

chpmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE CHPMV( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
COMPLEX ALPHA, BETA
COMPLEX A(*), X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE CHPMV_64( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
COMPLEX ALPHA, BETA
COMPLEX A(*), X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE HPMV( UPLO, [N], ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: A, X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE HPMV_64( UPLO, [N], ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: A, X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void chpmv(char uplo, int n, complex alpha, complex *a, complex *x, int incx, complex beta, complex *y, int incy);

void chpmv_64(char uplo, long n, complex alpha, complex *a, complex *x, long incx, complex beta, complex *y, long incy);

PURPOSE

chpmv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

chpr - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

SYNOPSIS

```
SUBROUTINE CHPR( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
COMPLEX X(*), A(*)
INTEGER N, INCX
REAL ALPHA

SUBROUTINE CHPR_64( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
COMPLEX X(*), A(*)
INTEGER*8 N, INCX
REAL ALPHA
```

F95 INTERFACE

```
SUBROUTINE HPR( UPLO, [N], ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: X, A
INTEGER :: N, INCX
REAL :: ALPHA

SUBROUTINE HPR_64( UPLO, [N], ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: X, A
INTEGER(8) :: N, INCX
REAL :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void chpr(char uplo, int n, float alpha, complex *x, int incx, complex *a);

void chpr_64(char uplo, long n, float alpha, complex *x, long incx, complex *a);

PURPOSE

chpr performs the hermitian rank 1 operation A := alpha*x*conjg(x') + A where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

NAME

chpr2 - perform the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

SYNOPSIS

```
SUBROUTINE CHPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
COMPLEX ALPHA
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY
 A

SUBROUTINE CHPR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
COMPLEX ALPHA
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
 A
```

F95 INTERFACE

```
SUBROUTINE HPR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY
 :: A

SUBROUTINE HPR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
 :: A
```

C INTERFACE

#include <sunperf.h>

void chpr2(char uplo, int n, complex alpha, complex *x, int incx, complex *y, int incy, a);

void chpr2_64(char uplo, long n, complex alpha, complex *x, long incx, complex *y, long incy, a);

PURPOSE

chpr2 performs the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

NAME

chprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CHPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CHPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPRFS( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HPRFS_64( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void chprfs(char uplo, int n, int nrhs, complex *a, complex *af, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void chprfs_64(char uplo, long n, long nrhs, complex *a, complex *af, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

chprfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CHPTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

chpsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CHPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPSV( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPSV_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chpsv(char uplo, int n, int nrhs, complex *a, int *ipivot, complex *b, int ldb, int *info);

void chpsv_64(char uplo, long n, long nrhs, complex *a, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

chpsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**H,  if UPLO = 'U', or

A = L * D * L**H,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CHPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = conjg(aji))

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

chpsvx - use the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE CHPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CHPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPSVX( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

```
 SUBROUTINE HPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void chpsvx(char fact, char uplo, int n, int nrhs, complex *a, complex *af, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void chpsvx_64(char fact, char uplo, long n, long nrhs, complex *a, complex *af, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

chpsvx uses the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**H, if UPLO = 'U', or

```
    A = L * D * L**H,  if UPLO = 'L',

 where U (or L) is a product of permutation and unit upper (lower)
 triangular matrices and D is Hermitian and block diagonal with
 1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CHPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CHPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as

reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

chptrd - reduce a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
SUBROUTINE CHPTRD( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), TAU(*)
INTEGER N, INFO
REAL D(*), E(*)

SUBROUTINE CHPTRD_64( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), TAU(*)
INTEGER*8 N, INFO
REAL D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRD( UPLO, [N], AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, TAU
INTEGER :: N, INFO
REAL, DIMENSION(:) :: D, E

SUBROUTINE HPTRD_64( UPLO, [N], AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, TAU
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void chptrd(char uplo, int n, complex *ap, float *d, float *e, complex *tau, int *info);

void chptrd_64(char uplo, long n, complex *ap, float *d, float *e, complex *tau, long *info);

PURPOSE

chptrd reduces a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in AP, overwriting A(1:i-1,i+1), and tau is stored in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in AP, overwriting A(i+2:n,i), and tau is stored in TAU(i).

NAME

chptrf - compute the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CHPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chptrf(char uplo, int n, complex *a, int *ipivot, int *info);

void chptrf_64(char uplo, long n, complex *a, long *ipivot, long *info);

PURPOSE

chptrf computes the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**H  or  A = L*D*L**H
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

chptri - compute the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE CHPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRI( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRI_64( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chptri(char uplo, int n, complex *a, int *ipivot, int *info);

void chptri_64(char uplo, long n, complex *a, long *ipivot, long *info);

PURPOSE

chptri computes the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (Hermitian) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

chptrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE CHPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CHPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRS( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRS_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void chptrs(char uplo, int n, int nrhs, complex *a, int *ipivot, complex *b, int ldb, int *info);

void chptrs_64(char uplo, long n, long nrhs, complex *a, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

chptrs solves a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

chsein - use inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H

SYNOPSIS

```
 SUBROUTINE CHSEIN( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, W, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 COMPLEX H(LDH,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER IFAILL(*), IFAILR(*)
 LOGICAL SELECT(*)
 REAL RWORK(*)

 SUBROUTINE CHSEIN_64( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, W, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 COMPLEX H(LDH,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER*8 IFAILL(*), IFAILR(*)
 LOGICAL*8 SELECT(*)
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEIN( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], W, VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: H, VL, VR
 INTEGER :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER, DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE HSEIN_64( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], W,
```

```
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], IFAILL, IFAILR,
*       [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: H, VL, VR
 INTEGER(8) :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER(8), DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void chsein(char side, char eigsrc, char initv, logical *select, int n, complex *h, int ldh, complex *w, complex *vl, int ldvl, complex *vr, int ldvr, int mm, int *m, int *ifaill, int *ifailr, int *info);

void chsein_64(char side, char eigsrc, char initv, logical *select, long n, complex *h, long ldh, complex *w, complex *vl, long ldvl, complex *vr, long ldvr, long mm, long *m, long *ifaill, long *ifailr, long *info);

PURPOSE

chsein uses inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H.

The right eigenvector x and the left eigenvector y of the matrix H corresponding to an eigenvalue w are defined by:

```
H * x = w * x,     y**h * H = w * y**h
```

where y**h denotes the conjugate transpose of the vector y.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R': compute right eigenvectors only;

  = 'L': compute left eigenvectors only;

  = 'B': compute both right and left eigenvectors.
  ```

- **EIGSRC (input)**
  Specifies the source of eigenvalues supplied in W:

  ```
  = 'Q': the eigenvalues were found using CHSEQR; thus, if
  H has zero subdiagonal elements, and so is
  block-triangular, then the j-th eigenvalue can be
  assumed to be an eigenvalue of the block containing
  the j-th row/column.  This property allows CHSEIN to
  perform inverse iteration on just one diagonal block.
  = 'N': no assumptions are made on the correspondence
  ```

```
between eigenvalues and diagonal blocks.  In this
case, CHSEIN must always perform inverse iteration
using the whole matrix H.
```

- **INITV (input)**

```
 = 'N': no initial vectors are supplied;

 = 'U': user-supplied initial vectors are stored in the arrays
VL and/or VR.
```

- **SELECT (input)**
  Specifies the eigenvectors to be computed. To select the eigenvector corresponding to the eigenvalue W(j), `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrix H. N >= 0.
- **H (input)**
  The upper Hessenberg matrix H.
- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **W (input/output)**
  On entry, the eigenvalues of H. On exit, the real parts of W may have been altered since close eigenvalues are perturbed slightly in searching for independent eigenvectors.
- **VL (input/output)**
  On entry, if INITV = 'U' and SIDE = 'L' or 'B', VL must contain starting vectors for the inverse iteration for the left eigenvectors; the starting vector for each eigenvector must be in the same column in which the eigenvector will be stored. On exit, if SIDE = 'L' or 'B', the left eigenvectors specified by SELECT will be stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if INITV = 'U' and SIDE = 'R' or 'B', VR must contain starting vectors for the inverse iteration for the right eigenvectors; the starting vector for each eigenvector must be in the same column in which the eigenvector will be stored. On exit, if SIDE = 'R' or 'B', the right eigenvectors specified by SELECT will be stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR required to store the eigenvectors (= the number of .TRUE. elements in SELECT).
- **WORK (workspace)**
  `dimension(N*N)`
- **RWORK (workspace)**
  `dimension(N)`
- **IFAILL (output)**
  If SIDE = 'L' or 'B', `IFAILL(i)` = j > 0 if the left eigenvector in the i-th column of VL (corresponding to the eigenvalue `w(j))` failed to converge; `IFAILL(i)` = 0 if the eigenvector converged satisfactorily. If SIDE = 'R', IFAILL is not referenced.
- **IFAILR (output)**
  If SIDE = 'R' or 'B', `IFAILR(i)` = j > 0 if the right eigenvector in the i-th column of VR (corresponding to the eigenvalue `w(j))` failed to converge; `IFAILR(i)` = 0 if the eigenvector converged satisfactorily. If SIDE = 'L', IFAILR is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, i is the number of eigenvectors which
failed to converge; see IFAILL and IFAILR for further
details.
```

FURTHER DETAILS

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x|+|y|.

NAME

chseqr - compute the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors

---

SYNOPSIS

```
 SUBROUTINE CHSEQR( JOB, COMPZ, N, ILO, IHI, H, LDH, W, Z, LDZ, WORK,
*      LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 COMPLEX H(LDH,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, ILO, IHI, LDH, LDZ, LWORK, INFO

 SUBROUTINE CHSEQR_64( JOB, COMPZ, N, ILO, IHI, H, LDH, W, Z, LDZ,
*      WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 COMPLEX H(LDH,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, ILO, IHI, LDH, LDZ, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE HSEQR( JOB, COMPZ, N, ILO, IHI, H, [LDH], W, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: H, Z
 INTEGER :: N, ILO, IHI, LDH, LDZ, LWORK, INFO

 SUBROUTINE HSEQR_64( JOB, COMPZ, N, ILO, IHI, H, [LDH], W, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: H, Z
 INTEGER(8) :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void chseqr(char job, char compz, int n, int ilo, int ihi, complex *h, int ldh, complex *w, complex *z, int ldz, int *info);

void chseqr_64(char job, char compz, long n, long ilo, long ihi, complex *h, long ldh, complex *w, complex *z, long ldz, long *info);

PURPOSE

chseqr computes the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input unitary matrix Q, so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the unitary matrix Q: A = Q*H*Q**H = (QZ)*T*(QZ)**H.

ARGUMENTS

- **JOB (input)**

```
  = 'E': compute eigenvalues only;

  = 'S': compute eigenvalues and the Schur form T.
```

- **COMPZ (input)**

```
  = 'N': no Schur vectors are computed;

  = 'I': Z is initialized to the unit matrix and the matrix Z
of Schur vectors of H is returned;
  = 'V': Z must contain an unitary matrix Q on entry, and
the product Q*Z is returned.
```

- **N (input)**
  The order of the matrix H. N > = 0.
- **ILO (input)**
  It is assumed that H is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGEBAL, and then passed to CGEHRD when the matrix output by CGEBAL is reduced to Hessenberg form. Otherwise ILO and IHI should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **H (input/output)**
  On entry, the upper Hessenberg matrix H. On exit, if JOB = 'S', H contains the upper triangular matrix T from the Schur decomposition (the Schur form). If JOB = 'E', the contents of H are unspecified on exit.
- **LDH (input)**
  The leading dimension of the array H. LDH > = max(1,N).

- **W (output)**
  The computed eigenvalues. If JOB = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in H, with `W(i)` = H(i,i).

- **Z (input/output)**
  If COMPZ = 'N': Z is not referenced.

  If COMPZ = 'I': on entry, Z need not be set, and on exit, Z contains the unitary matrix Z of the Schur vectors of H. If COMPZ = 'V': on entry Z must contain an N-by-N matrix Q, which is assumed to be equal to the unit matrix except for the submatrix Z(ILO:IHI,ILO:IHI); on exit Z contains Q*Z. Normally Q is the unitary matrix generated by CUNGHR after the call to CGEHRD which formed the Hessenberg matrix H.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= `max(1,N)` if COMPZ = 'I' or 'V'; LDZ >= 1 otherwise.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, CHSEQR failed to compute all the
eigenvalues in a total of 30*(IHI-ILO+1) iterations;
elements 1:ilo-1 and i+1:n of W contain those
eigenvalues which have been successfully computed.
```

NAME

jadmm, sjadmm, djadmm, cjadmm, zjadmm - Jagged diagonal matrix-matrix multiply (modified Ellpack)

SYNOPSIS

```
 SUBROUTINE SJADMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DJADMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CJADMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZJADMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

```
where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in jagged-diagonal format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

M             Number of rows in matrix A

N             Number of columns in matrix C

K             Number of columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices
```

VAL()	array of length NNZ consisting of entries of A. VAL can be viewed as a column major ordering of a row permutation of the Ellpack representation of A, where the Ellpack representation is permuted so that the rows are non-increasing in the number of nonzero entries. Values added for padding in Ellpack are not included in the Jagged-Diagonal format.
INDX()	array of length NNZ consisting of the column indices of the corresponding entries in VAL.
PNTR()	array of length MAXNZ+1, where PNTR(I)-PNTR(1)+1 points to the location in VAL of the first element in the row-permuted Ellpack represenation of A.
MAXNZ	max number of nonzeros elements per row.
IPERM()	integer array of length M such that I = IPERM(I'), where row I in the original Ellpack representation corresponds to row I' in the permuted representation. If IPERM(1) = 0, it is assumed by convention that IPERM(I) = I. IPERM is used to determine the order in which rows of C are updated.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.
LWORK	length of WORK array. LWORK is not referenced in the current version.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

jadrp, sjadrp, djadrp, cjadrp, zjadrp - right permutation of a jagged diagonal matrix

---

SYNOPSIS

```
 SUBROUTINE SJADRP( TRANSP, M, K, VAL, INDX, PNTR, MAXNZ,
*                   IPERM, WORK, LWORK )
 INTEGER*4  TRANSP, M, K, MAXNZ, LWORK
 INTEGER*4  INDX(*), PNTR(MAXNZ+1), IPERM(K)
 REAL*4     VAL(*), WORK(LWORK)

 SUBROUTINE DJADRP( TRANSP, M, K, VAL, INDX, PNTR, MAXNZ,
*                     IPERM,WORK,LWORK)
 INTEGER*4  TRANSP, M, K, MAXNZ, LWORK
 INTEGER*4  INDX(*), PNTR(MAXNZ+1), IPERM(K)
 REAL*8     VAL(*), WORK(LWORK)

 SUBROUTINE CJADRP( TRANSP, M, K, VAL, INDX, PNTR, MAXNZ,
*                   IPERM, WORK, LWORK )
 INTEGER*4  TRANSP, M, K, MAXNZ, LWORK
 INTEGER*4  INDX(*), PNTR(MAXNZ+1), IPERM(K)
 COMPLEX*8  VAL(*), WORK(LWORK)

 SUBROUTINE ZJADRP( TRANSP, M, K, VAL, INDX, PNTR, MAXNZ,
*                     IPERM,WORK,LWORK)
 INTEGER*4  TRANSP, M, K, MAXNZ, LWORK
 INTEGER*4  INDX(*), PNTR(MAXNZ+1), IPERM(K)
 COMPLEX*16 VAL(*), WORK(LWORK)
```

DESCRIPTION

```
A <- A P
A <- A P'
                                    ( ' indicates matrix transpose)
```

where permutation P is represented by an integer vector IPERM, such that IPERM(I) is equal to the position of the only nonzero element in row I of permutation matrix P.

NOTE: In order to get a symetrically permuted jagged diagonal matrix P A P', one can explicitly permute the columns P A by calling

```
   SJADRP(0, M, M, VAL, INDX, PNTR, MAXNZ, IPERM, WORK, LWORK)
```

where parameters VAL, INDX, PNTR, MAXNZ, IPERM are the representation of A in the jagged diagonal format. The operation makes sense if the original matrix A is square.

ARGUMENTS

TRANSP	Indicates how to operate with the permutation matrix 0 : operate with matrix 1 : operate with transpose matrix
M	Number of rows in matrix A
K	Number of columns in matrix A
VAL()	array of length PNTR(MAXNZ+1)-PNTR(1) consisting of entries of A. VAL can be viewed as a column major ordering of a row permutation of the Ellpack representation of A, where the Ellpack representation is permuted so that the rows are non-increasing in the number of nonzero entries. Values added for padding in Ellpack are not included in the Jagged-Diagonal format.
INDX()	array of length PNTR(MAXNZ+1)-PNTR(1) consisting of the column indices of the corresponding entries in VAL.
PNTR()	array of length MAXNZ+1, where PNTR(I)-PNTR(1)+1 points to the location in VAL of the first element in the row-permuted Ellpack represenation of A.
MAXNZ	max number of nonzeros elements per row.

```
IPERM()         integer array of length K such that I = IPERM(I').
                Array IPERM represents a permutation P, such that
                IPERM(I) is equal to the position of the only nonzero
                element in row I of permutation matrix P.
                For example, if
                              | 0 0 1 |
                         P  =| 1 0 0 |
                              | 0 1 0 |
                then IPERM = (3, 2, 1).

WORK()          scratch array of length LWORK.  LWORK should be at
                least K.

LWORK           length of WORK array

                If LWORK = -1, then a workspace query is assumed;
                the routine only calculates the optimal size of the
                WORK array, returns this value as the first entry of
                the WORK array, and no error message related to LWORK
                is issued by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NAME

jadsm, sjadsm, djadsm, cjadsm, zjadsm - Jagged-diagonal format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SJADSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DJADSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CJADSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZJADSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, PNTR, MAXNZ, IPERM,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5), MAXNZ,
*           LDB, LDC, LWORK
 INTEGER*4  INDX(NNZ), PNTR(MAXNZ+1), IPERM(M)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a diagonal scaling matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in jagged-diagonal format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
M	Number of rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
DV()	Array of length M containing the diagonal entries of the scaling diagonal matrix D.
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A') 5 : Diagonal 6 : Skew-Hermitian (A= -CONJG(A')) Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
  1 : lower
  2 : upper
DESCRA(3) main diagonal type
  0 : non-unit
  1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
  0 : C/C++ compatible
  1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
  0 : unknown
  1 : no repeated indices

VAL() array of length NNZ consisting of entries of A. VAL can be viewed as a column major ordering of a row permutation of the Ellpack representation of A, where the Ellpack representation is permuted so that the rows are non-increasing in the number of nonzero entries. Values added for padding in Ellpack are not included in the Jagged-Diagonal format.

INDX() array of length NNZ consisting of the column indices of the corresponding entries in VAL.

PNTR() array of length MAXNZ+1, where PNTR(I)-PNTR(1)+1 points to the location in VAL of the first element in the row-permuted Ellpack represenation of A.

MAXNZ max number of nonzeros elements per row.

IPERM() integer array of length M such that I = IPERM(I'), where row I in the original Ellpack representation corresponds to row I' in the permuted representation. If IPERM(1)=0, it's assumed by convention that IPERM(I)=I. IPERM is used to determine the order in which rows of C are updated.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least 2*M.

For good performance, LWORK should generally be larger.
For optimum performance on multiple processors, LWORK

```
>=2*M*N_CPUS where N_CPUS is the maximum number of
processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.


If LWORK = -1, then a workspace query is assumed; the
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

clarz - applie a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right

---

SYNOPSIS

```
SUBROUTINE CLARZ( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
COMPLEX TAU
COMPLEX V(*), C(LDC,*), WORK(*)
INTEGER M, N, L, INCV, LDC

SUBROUTINE CLARZ_64( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
COMPLEX TAU
COMPLEX V(*), C(LDC,*), WORK(*)
INTEGER*8 M, N, L, INCV, LDC
```

F95 INTERFACE

```
SUBROUTINE LARZ( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC], [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX :: TAU
COMPLEX, DIMENSION(:) :: V, WORK
COMPLEX, DIMENSION(:,:) :: C
INTEGER :: M, N, L, INCV, LDC

SUBROUTINE LARZ_64( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX :: TAU
COMPLEX, DIMENSION(:) :: V, WORK
COMPLEX, DIMENSION(:,:) :: C
INTEGER(8) :: M, N, L, INCV, LDC
```

C INTERFACE

#include <sunperf.h>

void clarz(char side, int m, int n, int l, complex *v, int incv, complex tau, complex *c, int ldc);

void clarz_64(char side, long m, long n, long l, complex *v, long incv, complex tau, complex *c, long ldc);

PURPOSE

clarz applies a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right. H is represented in the form

```
H = I - tau * v * v'
```

where tau is a complex scalar and v is a complex vector.

If tau = 0, then H is taken to be the unit matrix.

To apply H' (the conjugate transpose of H), supply `conjg(tau)` instead tau.

H is a product of k elementary reflectors as returned by CTZRZF.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form  H * C

  = 'R': form  C * H
  ```
- **M (input)**
  The number of rows of the matrix C.
- **N (input)**
  The number of columns of the matrix C.
- **L (input)**
  The number of entries of the vector V containing the meaningful part of the Householder vectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  The vector v in the representation of H as returned by CTZRZF. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0.
- **TAU (input)**
  The value tau in the representation of H.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by the matrix H * C if SIDE = 'L', or C * H if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).

- **WORK (workspace)**
  (N) if SIDE = 'L' or (M) if SIDE = 'R'

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

clarzb - applie a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right

---

SYNOPSIS

```
 SUBROUTINE CLARZB( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V, LDV,
*      T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 COMPLEX V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
 INTEGER M, N, K, L, LDV, LDT, LDC, LDWORK

 SUBROUTINE CLARZB_64( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V,
*      LDV, T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 COMPLEX V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
 INTEGER*8 M, N, K, L, LDV, LDT, LDC, LDWORK
```

F95 INTERFACE

```
 SUBROUTINE LARZB( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 COMPLEX, DIMENSION(:,:) :: V, T, C, WORK
 INTEGER :: M, N, K, L, LDV, LDT, LDC, LDWORK

 SUBROUTINE LARZB_64( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 COMPLEX, DIMENSION(:,:) :: V, T, C, WORK
 INTEGER(8) :: M, N, K, L, LDV, LDT, LDC, LDWORK
```

C INTERFACE

#include <sunperf.h>

void clarzb(char side, char trans, char direct, char storev, int m, int n, int k, int l, complex *v, int ldv, complex *t, int ldt, complex *c, int ldc, int ldwork);

void clarzb_64(char side, char trans, char direct, char storev, long m, long n, long k, long l, complex *v, long ldv, complex *t, long ldt, complex *c, long ldc, long ldwork);

PURPOSE

clarzb applies a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right.

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **SIDE (input)**

  = 'L': apply H or H' from the Left

  = 'R': apply H or H' from the Right

- **TRANS (input)**

  = 'N': apply H (No transpose)

  = 'C': apply H' (Conjugate transpose)

- **DIRECT (input)**
  Indicates how H is formed from a product of elementary reflectors = 'F': H = `H(1) H(2) . . . H(k)` (Forward, not supported yet)

  = 'B': H = H(k) . . . H(2) H(1) (Backward)

- **STOREV (input)**
  Indicates how the vectors which define the elementary reflectors are stored:

  = 'C': Columnwise (not supported yet)

  = 'R': Rowwise

- **M (input)**
  The number of rows of the matrix C.

- **N (input)**
  The number of columns of the matrix C.

- **K (input)**
  The order of the matrix T (= the number of elementary reflectors whose product defines the block reflector).

- **L (input)**
  The number of columns of the matrix V containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  If STOREV = 'C', NV = K; if STOREV = 'R', NV = L.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = L; if STOREV = 'R', LDV > = K.
- **T (input)**
  The triangular K-by-K matrix T in the representation of the block reflector.
- **LDT (input)**
  The leading dimension of the array T. LDT > = K.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by H*C or H'*C or C*H or C*H'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  `dimension(MAX(M,N),K)`
- **LDWORK (input)**
  The leading dimension of the array WORK. If SIDE = 'L', LDWORK > = max(1,N); if SIDE = 'R', LDWORK > = max(1,M).

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

clarzt - form the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors

---

SYNOPSIS

```
SUBROUTINE CLARZT( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
COMPLEX V(LDV,*), TAU(*), T(LDT,*)
INTEGER N, K, LDV, LDT

SUBROUTINE CLARZT_64( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
COMPLEX V(LDV,*), TAU(*), T(LDT,*)
INTEGER*8 N, K, LDV, LDT
```

F95 INTERFACE

```
SUBROUTINE LARZT( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
COMPLEX, DIMENSION(:) :: TAU
COMPLEX, DIMENSION(:,:) :: V, T
INTEGER :: N, K, LDV, LDT

SUBROUTINE LARZT_64( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
COMPLEX, DIMENSION(:) :: TAU
COMPLEX, DIMENSION(:,:) :: V, T
INTEGER(8) :: N, K, LDV, LDT
```

C INTERFACE

#include <sunperf.h>

void clarzt(char direct, char storev, int n, int k, complex *v, int ldv, complex *tau, complex *t, int ldt);

void clarzt_64(char direct, char storev, long n, long k, complex *v, long ldv, complex *tau, complex *t, long ldt);

PURPOSE

clarzt forms the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors.

If DIRECT = 'F', H = `H(1) H(2) . . . H(k)` and T is upper triangular;

If DIRECT = 'B', H = `H(k) . . . H(2) H(1)` and T is lower triangular.

If STOREV = 'C', the vector which defines the elementary reflector `H(i)` is stored in the i-th column of the array V, and

```
H  =  I - V * T * V'
```

If STOREV = 'R', the vector which defines the elementary reflector `H(i)` is stored in the i-th row of the array V, and

```
H  =  I - V' * T * V
```

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **DIRECT (input)**
  Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

  ```
  = 'F': H  = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H  = H(k) . . . H(2) H(1) (Backward)
  ```
- **STOREV (input)**
  Specifies how the vectors which define the elementary reflectors are stored (see also Further Details):

  ```
  = 'R': rowwise
  ```
- **N (input)**
  The order of the block reflector H. N > = 0.
- **K (input)**
  The order of the triangular factor T (= the number of elementary reflectors). K > = 1.
- **V (input)**
  (LDV,K) if STOREV = 'C' (LDV,N) if STOREV = 'R' The matrix V. See further details.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = max(1,N); if STOREV = 'R', LDV > = K.

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i).
- **T (output)**
  The k by k triangular factor T of the block reflector. If DIRECT = 'F', T is upper triangular; if DIRECT = 'B', T is lower triangular. The rest of the array is not used.
- **LDT (input)**
  The leading dimension of the array T. LDT >= K.

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The shape of the matrix V and the storage of the vectors which define the H(i) is best illustrated by the following example with n = 5 and k = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

DIRECT = 'F' and STOREV = 'C': DIRECT = 'F' and STOREV = 'R':

```
                                              ______V_____

     ( v1 v2 v3 )                            /           \
     ( v1 v2 v3 )                          ( v1 v1 v1 v1 v1 . . . . 1 )
V  = ( v1 v2 v3 )                           ( v2 v2 v2 v2 v2 . . . 1   )
     ( v1 v2 v3 )                          ( v3 v3 v3 v3 v3 . . 1     )
     ( v1 v2 v3 )

        .  .  .

        .  .  .

        1  .  .

           1  .

              1
```

DIRECT = 'B' and STOREV = 'C': DIRECT = 'B' and STOREV = 'R':

```
                                                    ______V_____
        1                                          /            \
        .  1                                    ( 1 . . . . v1 v1 v1 v1 v1 )
        .  .  1                                 ( . 1 . . . v2 v2 v2 v2 v2 )
        .  .  .                                 ( . . 1 . . v3 v3 v3 v3 v3 )
        .  .  .

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

```
V  = ( v1 v2 v3 )

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

NAME

clatzm - routine is deprecated and has been replaced by routine CUNMRZ

SYNOPSIS

```
SUBROUTINE CLATZM( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
COMPLEX TAU
COMPLEX V(*), C1(LDC,*), C2(LDC,*), WORK(*)
INTEGER M, N, INCV, LDC

SUBROUTINE CLATZM_64( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
COMPLEX TAU
COMPLEX V(*), C1(LDC,*), C2(LDC,*), WORK(*)
INTEGER*8 M, N, INCV, LDC
```

F95 INTERFACE

```
SUBROUTINE LATZM( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX :: TAU
COMPLEX, DIMENSION(:) :: V, WORK
COMPLEX, DIMENSION(:,:) :: C1, C2
INTEGER :: M, N, INCV, LDC

SUBROUTINE LATZM_64( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX :: TAU
COMPLEX, DIMENSION(:) :: V, WORK
COMPLEX, DIMENSION(:,:) :: C1, C2
INTEGER(8) :: M, N, INCV, LDC
```

C INTERFACE

#include <sunperf.h>

void clatzm(char side, int m, int n, complex *v, int incv, complex tau, complex *c1, complex *c2, int ldc);

void clatzm_64(char side, long m, long n, complex *v, long incv, complex tau, complex *c1, complex *c2, long ldc);

PURPOSE

clatzm routine is deprecated and has been replaced by routine CUNMRZ.

CLATZM applies a Householder matrix generated by CTZRQF to a matrix.

Let P = I - tau*u*u', u = (1),

```
                                ( v )
```

where v is an (m-1) vector if SIDE = 'L', or a (n-1) vector if SIDE = 'R'.

If SIDE equals 'L', let

```
       C = [ C1 ] 1
           [ C2 ] m-1
             n
```

Then C is overwritten by P*C.

If SIDE equals 'R', let

```
       C = [ C1, C2 ] m
             1  n-1
```

Then C is overwritten by C*P.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form P * C

  = 'R': form C * P
  ```

- **M (input)**
  The number of rows of the matrix C.
- **N (input)**

The number of columns of the matrix C.

- **V (input)**
  (1 + (M-1)*abs(INCV)) if SIDE = 'L' (1 + (N-1)*abs(INCV)) if SIDE = 'R' The vector v in the representation of P. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0
- **TAU (input)**
  The value tau in the representation of P.
- **C1 (input/output)**
  (LDC,N) if SIDE = 'L' (M,1) if SIDE = 'R' On entry, the n-vector C1 if SIDE = 'L', or the m-vector C1 if SIDE = 'R'.

  On exit, the first row of P*C if SIDE = 'L', or the first column of C*P if SIDE = 'R'.
- **C2 (input/output)**
  (LDC, N) if SIDE = 'L' (LDC, N-1) if SIDE = 'R' On entry, the (m - 1) x n matrix C2 if SIDE = 'L', or the m x (n - 1) matrix C2 if SIDE = 'R'.

  On exit, rows 2:m of P*C if SIDE = 'L', or columns 2:m of C*P if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the arrays C1 and C2. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'

NAME

cosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE COSQB( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE COSQB_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQB( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE COSQB_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void cosqb(int n, float *x, float *wsave);

void cosqb_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave cosine synthesis of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) that has been initialized by COSQI.

NAME

cosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE COSQF( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE COSQF_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQF( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE COSQF_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void cosqf(int n, float *x, float *wsave);

void cosqf_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave cosine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) that has been initialized by COSQI.

NAME

cosqi - initialize the array WSAVE, which is used in both COSQF and COSQB.

SYNOPSIS

```
SUBROUTINE COSQI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE COSQI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE COSQI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void cosqi(int n, float *wsave);

void cosqi_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. COSQI needs to be called only once to initialize WSAVE before calling COSQF and/or COSQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

cost - compute the discrete Fourier cosine transform of an even sequence. The COST transforms are unnormalized inverses of themselves, so a call of COST followed by another call of COST will multiply the input sequence by 2 * (N-1).

SYNOPSIS

```
SUBROUTINE COST( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE COST_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COST( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE COST_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void cost(int n, float *x, float *wsave);

void cost_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N - 1 is a product of small primes. N > = 2.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the cosine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by COSTI.

NAME

costi - initialize the array WSAVE, which is used in COST.

SYNOPSIS

```
SUBROUTINE COSTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE COSTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE COSTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void costi(int n, float *wsave);

void costi_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N - 1 is a product of small primes. N >
  = 2.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. COSTI is called once to initialize WSAVE before calling COST and need not be called again between calls to COST if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

cpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

SYNOPSIS

```
 SUBROUTINE CPBCON( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NDIAG, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)

 SUBROUTINE CPBCON_64( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBCON( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE PBCON_64( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cpbcon(char uplo, int n, int ndiag, complex *a, int lda, float anorm, float *rcond, int *info);

void cpbcon_64(char uplo, long n, long ndiag, complex *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

cpbcon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  `= 'U': Upper triangular factor stored in A;`

  `= 'L': Lower triangular factor stored in A.`
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian band matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**

dimension(N)

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

cpbequ - compute row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
 SUBROUTINE CPBEQU( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*)
 INTEGER N, NDIAG, LDA, INFO
 REAL SCOND, AMAX
 REAL SCALE(*)

 SUBROUTINE CPBEQU_64( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*)
 INTEGER*8 N, NDIAG, LDA, INFO
 REAL SCOND, AMAX
 REAL SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE PBEQU( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL :: SCOND, AMAX
 REAL, DIMENSION(:) :: SCALE

 SUBROUTINE PBEQU_64( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, NDIAG, LDA, INFO
 REAL :: SCOND, AMAX
 REAL, DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void cpbequ(char uplo, int n, int ndiag, complex *a, int lda, float *scale, float *scond, float *amax, int *info);

void cpbequ_64(char uplo, long n, long ndiag, complex *a, long lda, float *scale, float *scond, float *amax, long *info);

PURPOSE

cpbequ computes row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular of A is stored;

= 'L':  Lower triangular of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit
```

< 0: if INFO = -i, the i-th argument had an illegal value.

> 0: if INFO = i, the i-th diagonal element is nonpositive.

NAME

cpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CPBRFS( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPBRFS_64( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBRFS( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PBRFS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cpbrfs(char uplo, int n, int ndiag, int nrhs, complex *a, int lda, complex *af, int ldaf, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cpbrfs_64(char uplo, long n, long ndiag, long nrhs, complex *a, long lda, complex *af, long ldaf, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cpbrfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A as computed by CPBTRF, in the same storage format as A (see A).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPBTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cpbstf - compute a split Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE CPBSTF( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
COMPLEX AB(LDAB,*)
INTEGER N, KD, LDAB, INFO

SUBROUTINE CPBSTF_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
COMPLEX AB(LDAB,*)
INTEGER*8 N, KD, LDAB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBSTF( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: AB
INTEGER :: N, KD, LDAB, INFO

SUBROUTINE PBSTF_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: AB
INTEGER(8) :: N, KD, LDAB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpbstf(char uplo, int n, int kd, complex *ab, int ldab, int *info);

void cpbstf_64(char uplo, long n, long kd, complex *ab, long ldab, long *info);

PURPOSE

cpbstf computes a split Cholesky factorization of a complex Hermitian positive definite band matrix A.

This routine is designed to be used in conjunction with CHBGST.

The factorization has the form A = S**H*S where S is a band matrix of the same bandwidth as A and the following structure:

```
S = ( U    )

    ( M  L )
```

where U is upper triangular of order m = (n+kd)/2, and L is lower triangular of order n-m.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the factor S from the split Cholesky factorization A = S**H*S. See Further Details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD+1.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the factorization could not be completed,
because the updated element a(i,i) was negative; the
matrix A is not positive definite.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 7, KD = 2:

S = (s11 s12 s13)

```
    (       s22  s23  s24                )
    (            s33  s34                )
    (                 s44                )
    (            s53  s54  s55           )
    (                 s64  s65  s66      )
    (                      s75  s76  s77 )
```

If UPLO = 'U', the array AB holds:

on entry: on exit:

```
 *    *   a13  a24  a35  a46  a57   *    *   s13  s24  s53' s64' s75'
 *   a12  a23  a34  a45  a56  a67   *   s12  s23  s34  s54' s65' s76'
a11  a22  a33  a44  a55  a66  a77  s11  s22  s33  s44  s55  s66  s77
```

If UPLO = 'L', the array AB holds:

on entry: on exit:

a11 a22 a33 a44 a55 a66 a77 s11 s22 s33 s44 s55 s66 s77 a21 a32 a43 a54 a65 a76 * s12' s23' s34' s54 s65 s76 * a31 a42 a53 a64 a64 * * s13' s24' s53 s64 s75 * *

Array elements marked * are not used by the routine; s12' denotes conjg(s12); the diagonal elements of S are real.

NAME

cpbsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CPBSV( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE CPBSV_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBSV( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE PBSV_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpbsv(char uplo, int n, int ndiag, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void cpbsv_64(char uplo, long n, long ndiag, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

cpbsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite band matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H * U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular band matrix, and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

cpbsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CPBSVX( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPBSVX_64( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBSVX( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

```
 SUBROUTINE PBSVX_64( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cpbsvx(char fact, char uplo, int n, int ndiag, int nrhs, complex *a, int lda, complex *af, int ldaf, char equed, float *scale, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cpbsvx_64(char fact, char uplo, long n, long ndiag, long nrhs, complex *a, long lda, complex *af, long ldaf, char equed, float *scale, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cpbsvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite band matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**H * U, if UPLO = 'U', or

```
   A = L * L**H,  if UPLO = 'L',

where U is an upper triangular band matrix, and L is a lower
triangular band matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```
- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right-hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A (see A). If EQUED = 'Y', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'Y':  Equilibration was done, i.e., A has been replaced by
diag(SCALE) * A * diag(SCALE).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11  a12  a13
        a22  a23  a24
             a33  a34  a35
                  a44  a45  a46
                       a55  a56
   (aij =conjg(aji))         a66
```

Band storage of the upper triangle of A:

```
    *    *   a13  a24  a35  a46
    *   a12  a23  a34  a45  a56
   a11  a22  a33  a44  a55  a66
```

Similarly, if UPLO = 'L' the format of A is as follows:

```
   a11  a22  a33  a44  a55  a66
   a21  a32  a43  a54  a65   *
   a31  a42  a53  a64   *    *
```

Array elements marked * are not used by the routine.

NAME

cpbtf2 - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE CPBTF2( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
COMPLEX AB(LDAB,*)
INTEGER N, KD, LDAB, INFO

SUBROUTINE CPBTF2_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
COMPLEX AB(LDAB,*)
INTEGER*8 N, KD, LDAB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTF2( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: AB
INTEGER :: N, KD, LDAB, INFO

SUBROUTINE PBTF2_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: AB
INTEGER(8) :: N, KD, LDAB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpbtf2(char uplo, int n, int kd, complex *ab, int ldab, int *info);

void cpbtf2_64(char uplo, long n, long kd, complex *ab, long ldab, long *info);

PURPOSE

cpbtf2 computes the Cholesky factorization of a complex Hermitian positive definite band matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix, U' is the conjugate transpose of U, and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **KD (input)**
  The number of super-diagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. KD >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the band matrix A, in the same storage format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, KD = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

cpbtrf - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE CPBTRF( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, NDIAG, LDA, INFO

SUBROUTINE CPBTRF_64( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, NDIAG, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTRF( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INFO

SUBROUTINE PBTRF_64( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cpbtrf(char uplo, int n, int ndiag, complex *a, int lda, int *info);

void cpbtrf_64(char uplo, long n, long ndiag, complex *a, long lda, long *info);

PURPOSE

cpbtrf computes the Cholesky factorization of a complex Hermitian positive definite band matrix A.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the factorization could not be
completed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

Contributed by

Peter Mayes and Giuseppe Radicati, IBM ECSEC, Rome, March 23, 1989

NAME

cpbtrs - solve a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

SYNOPSIS

```
SUBROUTINE CPBTRS( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE CPBTRS_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTRS( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE PBTRS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpbtrs(char uplo, int n, int ndiag, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void cpbtrs_64(char uplo, long n, long ndiag, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

cpbtrs solves a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangular factor stored in A;

  = 'L':  Lower triangular factor stored in A.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

cpocon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
 SUBROUTINE CPOCON( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)

 SUBROUTINE CPOCON_64( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 REAL ANORM, RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE POCON( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE POCON_64( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cpocon(char uplo, int n, complex *a, int lda, float anorm, float *rcond, int *info);

void cpocon_64(char uplo, long n, complex *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

cpocon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cpoequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE CPOEQU( N, A, LDA, SCALE, SCOND, AMAX, INFO)
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
REAL SCOND, AMAX
REAL SCALE(*)

SUBROUTINE CPOEQU_64( N, A, LDA, SCALE, SCOND, AMAX, INFO)
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
REAL SCOND, AMAX
REAL SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE POEQU( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE

SUBROUTINE POEQU_64( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void cpoequ(int n, complex *a, int lda, float *scale, float *scond, float *amax, int *info);

void cpoequ_64(long n, complex *a, long lda, float *scale, float *scond, float *amax, long *info);

PURPOSE

cpoequ computes row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The N-by-N Hermitian positive definite matrix whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the i-th diagonal element is nonpositive.
```

NAME

cporfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite,

---

SYNOPSIS

```
 SUBROUTINE CPORFS( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPORFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PORFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PORFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B,
*        [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cporfs(char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cporfs_64(char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cporfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPOTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cposv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO

SUBROUTINE CPOSV_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE POSV( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO

SUBROUTINE POSV_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cposv(char uplo, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void cposv_64(char uplo, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

cposv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H* U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i of A is not
  positive definite, so the factorization could not be
  completed, and the solution has not been computed.
  ```

NAME

cposvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CPOSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*       SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPOSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*       SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE POSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2

 SUBROUTINE POSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, AF, B, X
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
REAL :: RCOND
REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cposvx(char fact, char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, char equed, float *scale, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cposvx_64(char fact, char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, char equed, float *scale, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cposvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**H* U, if UPLO = 'U', or

```
   A = L * L**H,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix diag(SCALE)*A*diag(SCALE).

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise,

SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS righthand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

cpotf2 - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

SYNOPSIS

```
SUBROUTINE CPOTF2( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE CPOTF2_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTF2( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTF2_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cpotf2(char uplo, int n, complex *a, int lda, int *info);

void cpotf2_64(char uplo, long n, complex *a, long lda, long *info);

PURPOSE

cpotf2 computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U'*U or A = L*L'.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

NAME

cpotrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

SYNOPSIS

```
SUBROUTINE CPOTRF( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE CPOTRF_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRF( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTRF_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cpotrf(char uplo, int n, complex *a, int lda, int *info);

void cpotrf_64(char uplo, long n, complex *a, long lda, long *info);

PURPOSE

cpotrf computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the block version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

NAME

cpotri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
SUBROUTINE CPOTRI( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE CPOTRI_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRI( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTRI_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cpotri(char uplo, int n, complex *a, int lda, int *info);

void cpotri_64(char uplo, long n, complex *a, long lda, long *info);

PURPOSE

cpotri computes the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF. On exit, the upper or lower triangle of the (Hermitian) inverse of A, overwriting the input factor U or L.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

cpotrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
SUBROUTINE CPOTRS( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO

SUBROUTINE CPOTRS_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRS( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO

SUBROUTINE POTRS_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpotrs(char uplo, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void cpotrs_64(char uplo, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

cpotrs solves a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cppcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE CPPCON( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER N, INFO
REAL ANORM, RCOND
REAL WORK2(*)

SUBROUTINE CPPCON_64( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
REAL ANORM, RCOND
REAL WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE PPCON( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2

SUBROUTINE PPCON_64( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void cppcon(char uplo, int n, complex *a, float anorm, float *rcond, int *info);

void cppcon_64(char uplo, long n, complex *a, float anorm, float *rcond, long *info);

PURPOSE

cppcon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cppequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE CPPEQU( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER N, INFO
REAL SCOND, AMAX
REAL SCALE(*)

SUBROUTINE CPPEQU_64( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER*8 N, INFO
REAL SCOND, AMAX
REAL SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE PPEQU( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE

SUBROUTINE PPEQU_64( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void cppequ(char uplo, int n, complex *a, float *scale, float *scond, float *amax, int *info);

void cppequ_64(char uplo, long n, complex *a, float *scale, float *scond, float *amax, long *info);

PURPOSE

cppequ computes row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). S contains the scale factors, S(i)=1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)=S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the i-th diagonal element is nonpositive.
  ```

NAME

cpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CPPRFS( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPPRFS_64( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PPRFS( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX], FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PPRFS_64( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX],
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cpprfs(char uplo, int n, int nrhs, complex *a, complex *af, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cpprfs_64(char uplo, long n, long nrhs, complex *a, complex *af, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cpprfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by SPPTRF/CPPTRF, packed columnwise in a linear array in the same format as A (see A).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cppsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CPPSV( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE CPPSV_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PPSV( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE PPSV_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cppsv(char uplo, int n, int nrhs, complex *a, complex *b, int ldb, int *info);

void cppsv_64(char uplo, long n, long nrhs, complex *a, complex *b, long ldb, long *info);

PURPOSE

cppsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H* U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

cppsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CPPSVX( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPPSVX_64( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PPSVX( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2

 SUBROUTINE PPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE,
*       B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
COMPLEX, DIMENSION(:) :: A, AF, WORK
COMPLEX, DIMENSION(:,:) :: B, X
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
REAL :: RCOND
REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cppsvx(char fact, char uplo, int n, int nrhs, complex *a, complex *af, char equed, float *scale, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cppsvx_64(char fact, char uplo, long n, long nrhs, complex *a, complex *af, char equed, float *scale, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cppsvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U'* U , if UPLO = 'U', or

```
   A = L * L',  if UPLO = 'L',

where U is an upper triangular matrix, L is a lower triangular
matrix, and ' indicates conjugate transpose.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

cpptrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE CPPTRF( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE CPPTRF_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRF( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE PPTRF_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void cpptrf(char uplo, int n, complex *a, int *info);

void cpptrf_64(char uplo, long n, complex *a, long *info);

PURPOSE

cpptrf computes the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

cpptri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE CPPTRI( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE CPPTRI_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRI( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE PPTRI_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void cpptri(char uplo, int n, complex *a, int *info);

void cpptri_64(char uplo, long n, complex *a, long *info);

PURPOSE

cpptri computes the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

ARGUMENTS

- **UPLO (input)**

```
  = 'U':  Upper triangular factor is stored in A;

  = 'L':  Lower triangular factor is stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise as a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.

  On exit, the upper or lower triangle of the (Hermitian) inverse of A, overwriting the input factor U or L.
- **INFO (output)**

```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

cpptrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE CPPTRS( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE CPPTRS_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRS( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE PPTRS_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void cpptrs(char uplo, int n, int nrhs, complex *a, complex *b, int ldb, int *info);

void cpptrs_64(char uplo, long n, long nrhs, complex *a, complex *b, long ldb, long *info);

PURPOSE

cpptrs solves a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

cptcon - compute the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF

SYNOPSIS

```
SUBROUTINE CPTCON( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
COMPLEX OFFD(*)
INTEGER N, INFO
REAL ANORM, RCOND
REAL DIAG(*), WORK(*)

SUBROUTINE CPTCON_64( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
COMPLEX OFFD(*)
INTEGER*8 N, INFO
REAL ANORM, RCOND
REAL DIAG(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTCON( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: OFFD
INTEGER :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: DIAG, WORK

SUBROUTINE PTCON_64( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: OFFD
INTEGER(8) :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: DIAG, WORK
```

C INTERFACE

#include <sunperf.h>

void cptcon(int n, float *diag, complex *offd, float anorm, float *rcond, int *info);

void cptcon_64(long n, float *diag, complex *offd, float anorm, float *rcond, long *info);

PURPOSE

cptcon computes the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF.

`Norm(inv(A))` is computed by a direct method, and the reciprocal of the condition number is computed as

```
RCOND = 1 / (ANORM * norm(inv(A))).
```

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization of A, as computed by CPTTRF.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization of A, as computed by CPTTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The method used is described in Nicholas J. Higham, ``Efficient Algorithms for Computing the Condition Number of a Tridiagonal Matrix'', SIAM J. Sci. Stat. Comput., Vol. 7, No. 1, January 1986.

NAME

cpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor

---

SYNOPSIS

```
SUBROUTINE CPTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
COMPLEX Z(LDZ,*)
INTEGER N, LDZ, INFO
REAL D(*), E(*), WORK(*)

SUBROUTINE CPTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
COMPLEX Z(LDZ,*)
INTEGER*8 N, LDZ, INFO
REAL D(*), E(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX, DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK

SUBROUTINE PTEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void cpteqr(char compz, int n, float *d, float *e, complex *z, int ldz, int *info);

void cpteqr_64(char compz, long n, float *d, float *e, complex *z, long ldz, long *info);

PURPOSE

cpteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor.

This routine computes the eigenvalues of the positive definite tridiagonal matrix to high relative accuracy. This means that if the eigenvalues range over many orders of magnitude in size, then the small eigenvalues and corresponding eigenvectors will be computed more accurately than, for example, with the standard QR method.

The eigenvectors of a full or band positive definite Hermitian matrix can also be found if CHETRD, CHPTRD, or CHBTRD has been used to reduce this matrix to tridiagonal form. (The reduction to tridiagonal form, however, may preclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix, if these eigenvalues range over many orders of magnitude.)

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvectors of original Hermitian
matrix also.  Array Z contains the unitary matrix
used to reduce the original matrix to tridiagonal
form.
 = 'I':  Compute eigenvectors of tridiagonal matrix also.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On normal exit, D contains the eigenvalues, in descending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', the unitary matrix used in the reduction to tridiagonal form. On exit, if COMPZ = 'V', the orthonormal eigenvectors of the original Hermitian matrix; if COMPZ = 'I', the orthonormal eigenvectors of the tridiagonal matrix. If INFO > 0 on exit, Z contains the eigenvectors associated with only the stored eigenvalues. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if COMPZ = 'V' or 'I', LDZ > = max(1,N).
- **WORK (workspace)**

dimension(4*N)

- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, and i is:
 < = N  the Cholesky factorization of the matrix could
not be performed because the i-th principal minor
was not positive definite.
 > N   the SVD algorithm failed to converge;
if INFO  = N+i, i off-diagonal elements of the
bidiagonal factor did not converge to zero.
```

NAME

cptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CPTRFS( UPLO, N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX OFFD(*), OFFDF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPTRFS_64( UPLO, N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX OFFD(*), OFFDF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PTRFS( UPLO, [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: OFFD, OFFDF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2

 SUBROUTINE PTRFS_64( UPLO, [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: OFFD, OFFDF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cptrfs(char uplo, int n, int nrhs, float *diag, complex *offd, float *diagf, complex *offdf, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void cptrfs_64(char uplo, long n, long nrhs, float *diag, complex *offd, float *diagf, complex *offdf, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

cptrfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:

  ```
   = 'U':  OFFD is the superdiagonal of A, and A  = U**H*DIAG*U;

   = 'L':  OFFD is the subdiagonal of A, and A  = L*DIAG*L**H.
  (The two forms are equivalent if A is real.)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n real diagonal elements of the tridiagonal matrix A.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the tridiagonal matrix A (see UPLO).
- **DIAGF (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization computed by CPTTRF.
- **OFFDF (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by CPTTRF (see UPLO).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

cptsv - compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

SYNOPSIS

```
SUBROUTINE CPTSV( N, NRHS, DIAG, SUB, B, LDB, INFO)
COMPLEX SUB(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
REAL DIAG(*)

SUBROUTINE CPTSV_64( N, NRHS, DIAG, SUB, B, LDB, INFO)
COMPLEX SUB(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
REAL DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTSV( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
COMPLEX, DIMENSION(:) :: SUB
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG

SUBROUTINE PTSV_64( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
COMPLEX, DIMENSION(:) :: SUB
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void cptsv(int n, int nrhs, float *diag, complex *sub, complex *b, int ldb, int *info);

void cptsv_64(long n, long nrhs, float *diag, complex *sub, complex *b, long ldb, long *info);

PURPOSE

cptsv computes the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

A is factored as A = L*D*L**H, and the factored form of A is then used to solve the system of equations.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the factorization A = L*DIAG*L**H.
- **SUB (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A. SUB can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U**H*DIAG*U factorization of A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the solution has not been
computed.  The factorization has not been completed
unless i  = N.
```

NAME

cptsvx - use the factorization A = L*D*L**H to compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE CPTSVX( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT
 COMPLEX SUB(*), SUBF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CPTSVX_64( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT
 COMPLEX SUB(*), SUBF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PTSVX( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT
 COMPLEX, DIMENSION(:) :: SUB, SUBF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2

 SUBROUTINE PTSVX_64( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT
 COMPLEX, DIMENSION(:) :: SUB, SUBF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
REAL :: RCOND
REAL, DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cptsvx(char fact, int n, int nrhs, float *diag, complex *sub, float *diagf, complex *subf, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cptsvx_64(char fact, long n, long nrhs, float *diag, complex *sub, float *diagf, complex *subf, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cptsvx uses the factorization A = L*D*L**H to compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the matrix A is factored as A = L*D*L**H, where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form

```
A = U**H*D*U.
```

2. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry. = 'F': On entry, DIAGF and SUBF contain the factored form of A. DIAG, SUB, DIAGF, and SUBF will not be modified. = 'N': The matrix A will be copied to DIAGF and SUBF and factored.
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.

- **SUB (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**H factorization of A. If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**H factorization of A.
- **SUBF (input/output)**
  If FACT = 'F', then SUBF is an input argument and on entry contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A. If FACT = 'N', then SUBF is an output argument and on exit contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

cpttrf - compute the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE CPTTRF( N, DIAG, OFFD, INFO)
COMPLEX OFFD(*)
INTEGER N, INFO
REAL DIAG(*)

SUBROUTINE CPTTRF_64( N, DIAG, OFFD, INFO)
COMPLEX OFFD(*)
INTEGER*8 N, INFO
REAL DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRF( [N], DIAG, OFFD, [INFO])
COMPLEX, DIMENSION(:) :: OFFD
INTEGER :: N, INFO
REAL, DIMENSION(:) :: DIAG

SUBROUTINE PTTRF_64( [N], DIAG, OFFD, [INFO])
COMPLEX, DIMENSION(:) :: OFFD
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void cpttrf(int n, float *diag, complex *offd, int *info);

void cpttrf_64(long n, float *diag, complex *offd, long *info);

PURPOSE

cpttrf computes the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A. The factorization may also be regarded as having the form A = U'*D*U.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U'*DIAG*U factorization of A.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, the leading minor of order k is not
positive definite; if k  < N, the factorization could not
be completed, while if k  = N, the factorization was
completed, but DIAG(N)  = 0.
```

NAME

cpttrs - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

SYNOPSIS

```
SUBROUTINE CPTTRS( UPLO, N, NRHS, DIAG, OFFD, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX OFFD(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
REAL DIAG(*)

SUBROUTINE CPTTRS_64( UPLO, N, NRHS, DIAG, OFFD, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX OFFD(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
REAL DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRS( UPLO, [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: OFFD
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG

SUBROUTINE PTTRS_64( UPLO, [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: OFFD
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void cpttrs(char uplo, int n, int nrhs, float *diag, complex *offd, complex *b, int ldb, int *info);

void cpttrs_64(char uplo, long n, long nrhs, float *diag, complex *offd, complex *b, long ldb, long *info);

PURPOSE

cpttrs solves a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF. D is a diagonal matrix specified in the vector D, U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **UPLO (input)**
  Specifies the form of the factorization and whether the vector OFFD is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L. = 'U': A = U'*DIAG*U, OFFD is the superdiagonal of U

  ```
  = 'L':  A  = L*DIAG*L', OFFD is the subdiagonal of L
  ```
- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization A = U'*DIAG*U or A = L*DIAG*L'.
- **OFFD (input/output)**
  If UPLO = 'U', the (n-1) superdiagonal elements of the unit bidiagonal factor U from the factorization A = U'*DIAG*U. If UPLO = 'L', the (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization A = L*DIAG*L'.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

cptts2 - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

SYNOPSIS

```
SUBROUTINE CPTTS2( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX E(*), B(LDB,*)
INTEGER IUPLO, N, NRHS, LDB
REAL D(*)

SUBROUTINE CPTTS2_64( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX E(*), B(LDB,*)
INTEGER*8 IUPLO, N, NRHS, LDB
REAL D(*)
```

F95 INTERFACE

```
SUBROUTINE CPTTS2( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX, DIMENSION(:) :: E
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: IUPLO, N, NRHS, LDB
REAL, DIMENSION(:) :: D

SUBROUTINE CPTTS2_64( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX, DIMENSION(:) :: E
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: IUPLO, N, NRHS, LDB
REAL, DIMENSION(:) :: D
```

C INTERFACE

#include <sunperf.h>

void cptts2(int iuplo, int n, int nrhs, float *d, complex *e, complex *b, int ldb);

void cptts2_64(long iuplo, long n, long nrhs, float *d, complex *e, complex *b, long ldb);

PURPOSE

cptts2 solves a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF. D is a diagonal matrix specified in the vector D, U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **IUPLO (input)**
  Specifies the form of the factorization and whether the vector E is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L. = 1: A = U'*D*U, E is the superdiagonal of U

  ```
  = 0:  A  = L*D*L', E is the subdiagonal of L
  ```
- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **D (input)**
  The n diagonal elements of the diagonal matrix D from the factorization A = U'*D*U or A = L*D*L'.
- **E (input)**
  If IUPLO = 1, the (n-1) superdiagonal elements of the unit bidiagonal factor U from the factorization A = U'*D*U.
  If IUPLO = 0, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization A = L*D*L'.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

NAME

crot - apply a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex

---

SYNOPSIS

```
SUBROUTINE CROT( N, X, INCX, Y, INCY, C, S)
COMPLEX S
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY
REAL C

SUBROUTINE CROT_64( N, X, INCX, Y, INCY, C, S)
COMPLEX S
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
REAL C
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX :: S
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY
REAL :: C

SUBROUTINE ROT_64( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX :: S
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
REAL :: C
```

C INTERFACE

#include <sunperf.h>

void crot(int n, complex *x, int incx, complex *y, int incy, float c, complex s);

void crot_64(long n, complex *x, long incx, complex *y, long incy, float c, complex s);

PURPOSE

crot applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex.

ARGUMENTS

- **N (input)**
  The number of elements in the vectors X and Y.
- **X (output)**
  On input, the vector X. On output, X is overwritten with C*X + S*Y.
- **INCX (input)**
  The increment between successive values of Y. INCX < > 0.
- **Y (output)**
  On input, the vector Y. On output, Y is overwritten with -CONJG(S)*X + C*Y.
- **INCY (input)**
  The increment between successive values of Y. INCY < > 0.
- **C (input)**
- **S (input)**
  .Vb 5 \& C and S define a rotation \& [C S] \& [-conjg(S) C] \& \& where C*C + S*CONJG(S) = 1.0. .Ve

---

NAME

crotg - Construct a Given's plane rotation

SYNOPSIS

```
SUBROUTINE CROTG( A, B, C, S)
COMPLEX A, B, S
REAL C

SUBROUTINE CROTG_64( A, B, C, S)
COMPLEX A, B, S
REAL C
```

F95 INTERFACE

```
SUBROUTINE ROTG( A, B, C, S)
COMPLEX :: A, B, S
REAL :: C

SUBROUTINE ROTG_64( A, B, C, S)
COMPLEX :: A, B, S
REAL :: C
```

C INTERFACE

#include <sunperf.h>

void crotg(complex *a, complex b, float *c, complex *s);

void crotg_64(complex *a, complex b, float *c, complex *s);

PURPOSE

crotg Construct a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **A (input/output)**
  On entry, A contains the entry in the first vector that corresponds to the element to be annihilated in the second vector. On exit, contains the nonzero element of the rotated vector.
- **B (input)**
  On entry, B contains the entry to be annihilated in the second vector. Unchanged on exit.
- **C (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.
- **S (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.

NAME

cscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE CSCAL( N, ALPHA, Y, INCY)
COMPLEX ALPHA
COMPLEX Y(*)
INTEGER N, INCY

SUBROUTINE CSCAL_64( N, ALPHA, Y, INCY)
COMPLEX ALPHA
COMPLEX Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: Y
INTEGER :: N, INCY

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
COMPLEX :: ALPHA
COMPLEX, DIMENSION(:) :: Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void cscal(int n, complex alpha, complex *y, int incy);

void cscal_64(long n, complex alpha, complex *y, long incy);

PURPOSE

cscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

csctr - Scatters elements from x into y.

SYNOPSIS

```
SUBROUTINE CSCTR(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE CSCTR_64(NZ, X, INDX, Y)

COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE SCTR([NZ], X, INDX, Y)

```
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE SCTR_64([NZ], X, INDX, Y)

COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

CSCTR - Scatters the components of a sparse vector x stored in compressed form into specified components of a vector y in full storage form.

```
do i = 1, n
  y(indx(i)) = x(i)
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values to be scattered from compressed form into full storage form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector whose elements specified by indx have been set to the corresponding entries of x. Only the elements corresponding to the indices in indx have been modified.

NAME

skymm, sskymm, dskymm, cskymm, zskymm - Skyline format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SSKYMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 REAL*4     ALPHA, BETA
 REAL*4     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DSKYMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 REAL*8     ALPHA, BETA
 REAL*8     VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CSKYMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DSKYMM( TRANSA, M, N, K, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, K, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

```
 where NNZ = PNTR(M+1)-PNTR(1) (upper triangular)
       NNZ = PNTR(K+1)-PNTR(1) (lower triangular)
       PNTR() size = (M+1) (upper triangular)
       PNTR() size = (K+1) (lower triangular)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C


where ALPHA and BETA are scalar, C and B are dense matrices,
A is a matrix represented in skyline format and
op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                    ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

M             Number of rows in matrix A

N             Number of columns in matrix C

K             Number of columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general (NOT SUPPORTED)
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
```

```
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
                1 : no repeated indices

VAL()         array contain the nonzeros of A in skyline profile form.
              Row-oriented if DESCRA(2) = 1 (lower triangular),
              column oriented if DESCRA(2) = 2 (upper triangular).

PNTR()        integer array of length M+1 (lower triangular) or
              K+1 (upper triangular) such that PNTR(I)-PNTR(1)+1
              points to the location in VAL of the first element of
              the skyline profile in row (column) I.

B()           rectangular array with first dimension LDB.

LDB           leading dimension of B

BETA          Scalar parameter

C()           rectangular array with first dimension LDC.

LDC           leading dimension of C

WORK()        scratch array of length LWORK. WORK is not
              referenced in the current version.


LWORK         length of WORK array. LWORK is not referenced
              in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

The SKY data structure is not supported for a general matrix structure (DESCRA(1)=0).

NAME

skysm, sskysm, dskysm, cskysm, zskysm - Skyline format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SSKYSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 REAL*4     ALPHA, BETA
 REAL*4     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DSKYSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 REAL*8     ALPHA, BETA
 REAL*8     DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CSKYSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZSKYSM( TRANSA, M, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, PNTR,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, M, N, UNITD, DESCRA(5),
*           LDB, LDC, LWORK
 INTEGER*4  PNTR(*),
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(M), VAL(NNZ), B(LDB,*), C(LDC,*), WORK(LWORK)
```

```
 where NNZ = PNTR(M+1)-PNTR(1) (upper triangular)
       NNZ = PNTR(K+1)-PNTR(1) (lower triangular)
       PNTR() size = (M+1) (upper triangular)
       PNTR() size = (K+1) (lower triangular)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices,
D is a diagonal scaling matrix, A is a unit, or non-unit, upper or
lower triangular matrix represented in skyline format and
op(A) is one of

```
 op( A ) = inv(A) or  op( A ) = inv(A')  or  op( A ) =inv(conjg( A' )).
 (inv denotes matrix inverse,  ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if matrix is real.

M             Number of rows in matrix A

N             Number of columns in matrix C

UNITD         Type of scaling:
                1 : Identity matrix (argument DV[] is ignored)
                2 : Scale on left (row scaling)
                3 : Scale on right (column scaling)

DV()          Array of length M containing the diagonal entries of the
              scaling diagonal matrix D.

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
```

4 : Skew(Anti)-Symmetric (A=-A')
5 : Diagonal
6 : Skew-Hermitian (A= -CONJG(A'))

Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-unit
1 : unit
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() array contain the nonzeros of A in skyline profile form. Row-oriented if DESCRA(2) = 1 (lower triangular), column oriented if DESCRA(2) = 2 (upper triangular).

PNTR() integer array of length M+1 (lower triangular) or K+1 (upper triangular) such that PNTR(I)-PNTR(1)+1 points to the location in VAL of the first element of the skyline profile in row (column) I.

B() rectangular array with first dimension LDB.

LDB leading dimension of B

BETA Scalar parameter

C() rectangular array with first dimension LDC.

LDC leading dimension of C

WORK() scratch array of length LWORK.
On exit, if LWORK = -1, WORK(1) returns the optimum LWORK.

LWORK length of WORK array. LWORK should be at least M.

For good performance, LWORK should generally be larger. For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program.

If LWORK=0, the routine is to allocate workspace needed.

If LWORK = -1, then a workspace query is assumed; the

```
routine only calculates the optimum size of the WORK
array, returns this value as the first entry of the WORK
array, and no error message related to LWORK is issued
by XERBLA.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

NAME

cspcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE CSPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)
REAL ANORM, RCOND

SUBROUTINE CSPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
REAL ANORM, RCOND
```

F95 INTERFACE

```
SUBROUTINE SPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND

SUBROUTINE SPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void cspcon(char uplo, int n, complex *a, int *ipivot, float anorm, float *rcond, int *info);

void cspcon_64(char uplo, long n, complex *a, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

cspcon estimates the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

csprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CSPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CSPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SPRFS( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SPRFS_64( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void csprfs(char uplo, int n, int nrhs, complex *a, complex *af, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void csprfs_64(char uplo, long n, long nrhs, complex *a, complex *af, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

csprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CSPTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cspsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE CSPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPSV( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPSV_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void cspsv(char uplo, int n, int nrhs, complex *a, int *ipivot, complex *b, int ldb, int *info);

void cspsv_64(char uplo, long n, long nrhs, complex *a, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

cspsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = aji)

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

cspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE CSPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CSPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SPSVX( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

```
 SUBROUTINE SPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: A, AF, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void cspsvx(char fact, char uplo, int n, int nrhs, complex *a, complex *af, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void cspsvx_64(char fact, char uplo, long n, long nrhs, complex *a, complex *af, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

cspsvx uses the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**T, if UPLO = 'U', or

```
    A = L * D * L**T,  if UPLO = 'L',

 where U (or L) is a product of permutation and unit upper (lower)
 triangular matrices and D is symmetric and block diagonal with
 1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CSPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the

largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

csptrf - compute the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CSPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csptrf(char uplo, int n, complex *a, int *ipivot, int *info);

void csptrf_64(char uplo, long n, complex *a, long *ipivot, long *info);

PURPOSE

csptrf computes the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

csptri - compute the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE CSPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRI( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRI_64( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csptri(char uplo, int n, complex *a, int *ipivot, int *info);

void csptri_64(char uplo, long n, complex *a, long *ipivot, long *info);

PURPOSE

csptri computes the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.

- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.

- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

csptrs - solve a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE CSPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRS( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRS_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csptrs(char uplo, int n, int nrhs, complex *a, int *ipivot, complex *b, int ldb, int *info);

void csptrs_64(char uplo, long n, long nrhs, complex *a, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

csptrs solves a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

csrot - Apply a plane rotation.

SYNOPSIS

```
SUBROUTINE CSROT( N, X, INCX, Y, INCY, C, S)
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY
REAL C, S

SUBROUTINE CSROT_64( N, X, INCX, Y, INCY, C, S)
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
REAL C, S
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY
REAL :: C, S

SUBROUTINE ROT_64( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
REAL :: C, S
```

C INTERFACE

#include <sunperf.h>

void csrot(int n, complex *x, int incx, complex *y, int incy, float c, float s);

void csrot_64(long n, complex *x, long incx, complex *y, long incy, float c, float s);

PURPOSE

csrot Apply a plane rotation, where the cos and sin (c and s) are real and the vectors x and y are complex.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (output)**
  On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **C (input)**
  On entry, the cosine. Unchanged on exit.
- **S (input)**
  On entry, the sin. Unchanged on exit.

NAME

csscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE CSSCAL( N, ALPHA, Y, INCY)
COMPLEX Y(*)
INTEGER N, INCY
REAL ALPHA

SUBROUTINE CSSCAL_64( N, ALPHA, Y, INCY)
COMPLEX Y(*)
INTEGER*8 N, INCY
REAL ALPHA
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
COMPLEX, DIMENSION(:) :: Y
INTEGER :: N, INCY
REAL :: ALPHA

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
COMPLEX, DIMENSION(:) :: Y
INTEGER(8) :: N, INCY
REAL :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void csscal(int n, float alpha, complex *y, int incy);

void csscal_64(long n, float alpha, complex *y, long incy);

PURPOSE

csscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

cstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

---

SYNOPSIS

```
 SUBROUTINE CSTEDC( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 COMPZ
 COMPLEX Z(LDZ,*), WORK(*)
 INTEGER N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL D(*), E(*), RWORK(*)

 SUBROUTINE CSTEDC_64( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 COMPZ
 COMPLEX Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL D(*), E(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEDC( COMPZ, [N], D, E, Z, [LDZ], WORK, [LWORK], RWORK,
*       [LRWORK], IWORK, [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: D, E, RWORK

 SUBROUTINE STEDC_64( COMPZ, [N], D, E, Z, [LDZ], WORK, [LWORK],
*       RWORK, [LRWORK], IWORK, [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
```

```
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL, DIMENSION(:) :: D, E, RWORK
```

C INTERFACE

#include <sunperf.h>

void cstedc(char compz, int n, float *d, float *e, complex *z, int ldz, complex *work, int lwork, float *rwork, int lrwork, int *iwork, int liwork, int *info);

void cstedc_64(char compz, long n, float *d, float *e, complex *z, long ldz, complex *work, long lwork, float *rwork, long lrwork, long *iwork, long liwork, long *info);

PURPOSE

cstedc computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band complex Hermitian matrix can also be found if CHETRD or CHPTRD or CHBTRD has been used to reduce this matrix to tridiagonal form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLAED3 for details.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'I':  Compute eigenvectors of tridiagonal matrix also.

 = 'V':  Compute eigenvectors of original Hermitian matrix
also.  On entry, Z contains the unitary matrix used
to reduce the original matrix to tridiagonal form.
```

- **N (input)**
  The dimension of the symmetric tridiagonal matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the unitary matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original Hermitian matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z

is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1. If eigenvectors are desired, then LDZ >= max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If COMPZ = 'N' or 'I', or N <= 1, LWORK must be at least 1. If COMPZ = 'V' and N > 1, LWORK must be at least N*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, RWORK(1) returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If COMPZ = 'N' or N <= 1, LRWORK must be at least 1. If COMPZ = 'V' and N > 1, LRWORK must be at least 1 + 3*N + 2*N*lg N + 3*N**2 , where lg(N) = smallest integer k such that 2**k >= N. If COMPZ = 'I' and N > 1, LRWORK must be at least 1 + 4*N + 2*N**2 .

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (output)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If COMPZ = 'N' or N <= 1, LIWORK must be at least 1. If COMPZ = 'V' or N > 1, LIWORK must be at least 6 + 6*N + 5*N*lg N. If COMPZ = 'I' or N > 1, LIWORK must be at least 3 + 5*N .

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an eigenvalue while
working on the submatrix lying in rows and columns
INFO/(N+1) through mod(INFO,N+1).
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

NAME

cstegr - Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

---

SYNOPSIS

```
 SUBROUTINE CSTEGR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*      W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 COMPLEX Z(LDZ,*)
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), WORK(*)

 SUBROUTINE CSTEGR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*      M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 COMPLEX Z(LDZ,*)
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEGR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEGR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL :: VL, VU, ABSTOL
REAL, DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void cstegr(char jobz, char range, int n, float *d, float *e, float vl, float vu, int il, int iu, float abstol, int *m, float *w, complex *z, int ldz, int *isuppz, int *info);

void cstegr_64(char jobz, char range, long n, float *d, float *e, float vl, float vu, long il, long iu, float abstol, long *m, float *w, complex *z, long ldz, long *isuppz, long *info);

PURPOSE

cstegr b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high relative accuracy by the dqds algorithm,

```
(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

Note 1 : Currently CSTEGR is only set up to find ALL the n eigenvalues and eigenvectors of T in `O(n^2)` time

Note 2 : Currently the routine CSTEIN is called when an appropriate sigma_i cannot be chosen in step (c) above. CSTEIN invokes modified Gram-Schmidt when eigenvalues are close.

Note 3 : CSTEGR works only on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs. Normal execution of CSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the ieee standard.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix T. On exit, D is overwritten.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix T in elements 1 to N-1 of E; `E(N)` need not be set. On exit, E is overwritten.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues/eigenvectors. IF JOBZ = 'V', the eigenvalues and eigenvectors output have residual norms bounded by ABSTOL, and the dot products between different eigenvectors are bounded by ABSTOL. If ABSTOL is less than N*EPS*|T|, then N*EPS*|T| will be used in its place, where EPS is the machine precision and |T| is the 1-norm of the tridiagonal matrix. The eigenvalues are computed to an accuracy of EPS*|T| irrespective of ABSTOL. If high relative accuracy is important, set ABSTOL to DLAMCH('Safe minimum'). See Barlow and Demmel ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7 for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**

The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= `max(1,18*N)`

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= `max(1,10*N)`

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = 1, internal error in SLARRE,
if INFO  = 2, internal error in CLARRV.
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA
```

NAME

cstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

---

SYNOPSIS

```
 SUBROUTINE CSTEIN( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 COMPLEX Z(LDZ,*)
 INTEGER N, M, LDZ, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 REAL D(*), E(*), W(*), WORK(*)

 SUBROUTINE CSTEIN_64( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 COMPLEX Z(LDZ,*)
 INTEGER*8 N, M, LDZ, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 REAL D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEIN( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER :: N, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL, DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEIN_64( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ],
*       [WORK], [IWORK], IFAIL, [INFO])
 COMPLEX, DIMENSION(:,:) :: Z
 INTEGER(8) :: N, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL, DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void cstein(int n, float *d, float *e, int m, float *w, int *iblock, int *isplit, complex *z, int ldz, int *ifail, int *info);

void cstein_64(long n, float *d, float *e, long m, float *w, long *iblock, long *isplit, complex *z, long ldz, long *ifail, long *info);

PURPOSE

cstein computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter MAXITS (currently set to 5).

Although the eigenvectors are real, they are stored in a complex array, which may be passed to CUNMTR or CUPMTR for back

transformation to the eigenvectors of a complex Hermitian matrix which was reduced to tridiagonal form.

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix T, stored in elements 1 to N-1; `E(N)` need not be set.
- **M (input)**
  The number of eigenvectors to be found. 0 < = M < = N.
- **W (input)**
  The first M elements of W contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array W from SSTEBZ with ORDER = 'B' is expected here.)
- **IBLOCK (input)**
  The submatrix indices associated with the corresponding eigenvalues in W; `IBLOCK(i)` =1 if eigenvalue `W(i)` belongs to the first submatrix from the top, =2 if `W(i)` belongs to the second submatrix, etc. (The output array IBLOCK from SSTEBZ is expected here.)
- **ISPLIT (input)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc. (The output array ISPLIT from SSTEBZ is expected here.)
- **Z (output)**
  The computed eigenvectors. The eigenvector associated with the eigenvalue `W(i)` is stored in the i-th column of Z. Any vector which fails to converge is set to its current iterate after MAXITS iterations. The imaginary parts of the eigenvectors are set to zero.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK (workspace)**
  `dimension(N)`
- **IFAIL (output)**
  On normal exit, all elements of IFAIL are zero. If one or more eigenvectors fail to converge after MAXITS iterations, then their indices are stored in array IFAIL.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, then i eigenvectors failed to converge
in MAXITS iterations.  Their indices are stored in
array IFAIL.
```

NAME

csteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

---

SYNOPSIS

```
SUBROUTINE CSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
COMPLEX Z(LDZ,*)
INTEGER N, LDZ, INFO
REAL D(*), E(*), WORK(*)

SUBROUTINE CSTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
COMPLEX Z(LDZ,*)
INTEGER*8 N, LDZ, INFO
REAL D(*), E(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX, DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK

SUBROUTINE STEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX, DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void csteqr(char compz, int n, float *d, float *e, complex *z, int ldz, int *info);

void csteqr_64(char compz, long n, float *d, float *e, complex *z, long ldz, long *info);

PURPOSE

csteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. The eigenvectors of a full or band complex Hermitian matrix can also be found if CHETRD or CHPTRD or CHBTRD has been used to reduce this matrix to tridiagonal form.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvalues and eigenvectors of the original
Hermitian matrix.  On entry, Z must contain the
unitary matrix used to reduce the original matrix
to tridiagonal form.
 = 'I':  Compute eigenvalues and eigenvectors of the
tridiagonal matrix.  Z is initialized to the identity
matrix.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the unitary matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original Hermitian matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if eigenvectors are desired, then LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(max(1,2*N-2))` If COMPZ = 'N', then WORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: the algorithm has failed to find all the eigenvalues in a total of 30*N iterations; if INFO = i, then i elements of E have not converged to zero; on exit, D and E contain the elements of a symmetric tridiagonal matrix which is unitarily similar to the original matrix.

NAME

cstsv - compute the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix

---

SYNOPSIS

```
SUBROUTINE CSTSV( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)

SUBROUTINE CSTSV_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STSV( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STSV_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void cstsv(int n, int nrhs, complex *l, complex *d, complex *subl, complex *b, int ldb, int *ipiv, int *info);

void cstsv_64(long n, long nrhs, complex *l, complex *d, complex *subl, complex *b, long ldb, long *ipiv, long *info);

PURPOSE

cstsv computes the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides in B.
- **L (input/output)**

```
COMPLEX array, dimension (N)
```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.
- **D (input/output)**

```
REAL array, dimension (N)
```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.
- **SUBL (output)**

```
COMPLEX array, dimension (N)
```

  On exit, part of the factorization of A.
- **B (input/output)**
  The columns of B contain the right hand sides.
- **LDB (input)**
  The leading dimension of B as specified in a type or DIMENSION statement.
- **IPIV (output)**

```
INTEGER array, dimension (N)
```

  On exit, the pivot indices of the factorization.
- **INFO (output)**

```
INTEGER

= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, D(k,k) is exactly zero.  The factorization has
been completed, but the block diagonal matrix D is exactly singular
```

and division by zero will occur if it is used to solve a system of equations.

NAME

csttrf - compute the factorization of a complex Hermitian tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE CSTTRF( N, L, D, SUBL, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*)
INTEGER N, INFO
INTEGER IPIV(*)

SUBROUTINE CSTTRF_64( N, L, D, SUBL, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*)
INTEGER*8 N, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STTRF( [N], L, D, SUBL, IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STTRF_64( [N], L, D, SUBL, IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void csttrf(int n, complex *l, complex *d, complex *subl, int *ipiv, int *info);

void csttrf_64(long n, complex *l, complex *d, complex *subl, long *ipiv, long *info);

PURPOSE

csttrf computes the L*D*L**H factorization of a complex Hermitian tridiagonal matrix A.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.

- **L (input/output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.

- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.

- **SUBL (output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On exit, part of the factorization of A.

- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.

- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, D(k,k) is exactly zero.  The factorization has
  been completed, but the block diagonal matrix D is exactly singular
  and division by zero will occur if it is used to solve a system of
  equations.
  ```

NAME

csttrs - computes the solution to a complex system of linear equations A * X = B

SYNOPSIS

```
SUBROUTINE CSTTRS( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)

SUBROUTINE CSTTRS_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STTRS( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STTRS_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX, DIMENSION(:) :: L, D, SUBL
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void csttrs(int n, int nrhs, complex *l, complex *d, complex *subl, complex *b, int ldb, int *ipiv, int *info);

void csttrs_64(long n, long nrhs, complex *l, complex *d, complex *subl, complex *b, long ldb, long *ipiv, long *info);

PURPOSE

csttrs computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric tridiagonal matrix and X and B are N-by-NRHS matrices.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.
- **NRHS (input)**

  ```
  INTEGER
  ```

  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **L (input)**

  ```
  COMPLEX array, dimension (N-1)
  ```

  On entry, the subdiagonal elements of LL and DD.
- **D (input)**

  ```
  COMPLEX array, dimension (N)
  ```

  On entry, the diagonal elements of DD.
- **SUBL (input)**

  ```
  COMPLEX array, dimension (N-2)
  ```

  On entry, the second subdiagonal elements of LL.
- **B (input)**

  ```
  COMPLEX array, dimension (LDB, NRHS)
  ```

  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**

  ```
  INTEGER
  ```

  The leading dimension of the array B. LDB > = max(1, N)
- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

Details of the interchanges and block pivot. If IPIV(K) > 0, 1 by 1 pivot, and if IPIV(K) = K + 1 an interchange done; If IPIV(K) < 0, 2 by 2 pivot, no interchange required.

- **INFO (output)**

```
INTEGER

= 0:  successful exit

< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

cswap - Exchange vectors x and y.

SYNOPSIS

```
SUBROUTINE CSWAP( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE CSWAP_64( N, X, INCX, Y, INCY)
COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE SWAP( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE SWAP_64( [N], X, [INCX], Y, [INCY])
COMPLEX, DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void cswap(int n, complex *x, int incx, complex *y, int incy);

void cswap_64(long n, complex *x, long incx, complex *y, long incy);

PURPOSE

cswap Exchange x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input/output)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, the y vector.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, the x vector.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

csycon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
 SUBROUTINE CSYCON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*)
 REAL ANORM, RCOND

 SUBROUTINE CSYCON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*)
 REAL ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE SYCON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: ANORM, RCOND

 SUBROUTINE SYCON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void csycon(char uplo, int n, complex *a, int lda, int *ipivot, float anorm, float *rcond, int *info);

void csycon_64(char uplo, long n, complex *a, long lda, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

csycon estimates the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

csymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE CSYMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, LDA, LDB, LDC

 SUBROUTINE CSYMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, LDA, LDB, LDC

 SUBROUTINE SYMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void csymm(char side, char uplo, int m, int n, complex alpha, complex *a, int lda, complex *b, int ldb, complex beta, complex *c, int ldc);

void csymm_64(char side, char uplo, long m, long n, complex alpha, complex *a, long lda, complex *b, long ldb, complex beta, complex *c, long ldc);

PURPOSE

csymm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the symmetric matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the symmetric matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the symmetric matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the symmetric matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the symmetric matrix, such that when

UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

csyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

SYNOPSIS

```
 SUBROUTINE CSYR2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER N, K, LDA, LDB, LDC

 SUBROUTINE CSYR2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYR2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: N, K, LDA, LDB, LDC

 SUBROUTINE SYR2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: N, K, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void csyr2k(char uplo, char transa, int n, int k, complex alpha, complex *a, int lda, complex *b, int ldb, complex beta, complex *c, int ldc);

void csyr2k_64(char uplo, char transa, long n, long k, complex alpha, complex *a, long lda, complex *b, long ldb, complex beta, complex *c, long ldc);

PURPOSE

csyr2k K performs one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*B' + alpha*B*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*B + alpha*B'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'T' or 't', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

csyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE CSYRFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CSYRFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SYRFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SYRFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void csyrfs(char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void csyrfs_64(char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

csyrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CSYTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

csyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

SYNOPSIS

```
 SUBROUTINE CSYRK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), C(LDC,*)
 INTEGER N, K, LDA, LDC

 SUBROUTINE CSYRK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 COMPLEX ALPHA, BETA
 COMPLEX A(LDA,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYRK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: N, K, LDA, LDC

 SUBROUTINE SYRK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX :: ALPHA, BETA
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: N, K, LDA, LDC
```

C INTERFACE

#include <sunperf.h>

void csyrk(char uplo, char transa, int n, int k, complex alpha, complex *a, int lda, complex beta, complex *c, int ldc);

void csyrk_64(char uplo, char transa, long n, long k, complex alpha, complex *a, long lda, complex beta, complex *c, long ldc);

PURPOSE

csyrk performs one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'T' or 't', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

csysv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CSYSV( UPLO, N, NRHS, A, LDA, IPIV, B, LDB, WORK, LWORK,
*      INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER IPIV(*)

 SUBROUTINE CSYSV_64( UPLO, N, NRHS, A, LDA, IPIV, B, LDB, WORK,
*      LWORK, INFO)
 CHARACTER * 1 UPLO
 COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSV( UPLO, [N], [NRHS], A, [LDA], IPIV, B, [LDB], [WORK],
*       [LWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER, DIMENSION(:) :: IPIV

 SUBROUTINE SYSV_64( UPLO, [N], [NRHS], A, [LDA], IPIV, B, [LDB],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void csysv(char uplo, int n, int nrhs, complex *a, int lda, int *ipiv, complex *b, int ldb, int *info);

void csysv_64(char uplo, long n, long nrhs, complex *a, long lda, long *ipiv, complex *b, long ldb, long *info);

PURPOSE

csysv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D, as determined by CSYTRF. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a

2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of WORK. LWORK >= 1, and for best performance LWORK >= N*NB, where NB is the optimal blocksize for CSYTRF.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

csysvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE CSYSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CSYSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
 REAL RCOND
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SYSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
```

```
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void csysvx(char fact, char uplo, int n, int nrhs, complex *a, int lda, complex *af, int ldaf, int *ipivot, complex *b, int ldb, complex *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void csysvx_64(char fact, char uplo, long n, long nrhs, complex *a, long lda, complex *af, long ldaf, long *ipivot, complex *b, long ldb, complex *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

csysvx uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
      A = U * D * U**T,  if UPLO = 'U', or

      A = L * D * L**T,  if UPLO = 'L',

   where U (or L) is a product of permutation and unit upper (lower)
   triangular matrices, and D is symmetric and block diagonal with
   1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CSYTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 2*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

csytf2 - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CSYTF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE CSYTF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE SYTF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE SYTF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void csytf2(char uplo, int n, complex *a, int lda, int *ipiv, int *info);

void csytf2_64(char uplo, long n, complex *a, long lda, long *ipiv, long *info);

PURPOSE

csytf2 computes the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the transpose of U, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

csytrf - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE CSYTRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSYTRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csytrf(char uplo, int n, complex *a, int lda, int *ipivot, int *info);

void csytrf_64(char uplo, long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

csytrf computes the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    v    0   )   k-s

U(k)  =  (   0    I    0   )   s

         (   0    0    I   )   n-k

            k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    0     0   )  k-1

L(k)  =  (   0    I     0   )  s
```

```
(   0    v     I   )  n-k-s+1

   k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

csytri - compute the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
SUBROUTINE CSYTRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSYTRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csytri(char uplo, int n, complex *a, int lda, int *ipivot, int *info);

void csytri_64(char uplo, long n, complex *a, long lda, long *ipivot, long *info);

PURPOSE

csytri computes the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

csytrs - solve a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
SUBROUTINE CSYTRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE CSYTRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void csytrs(char uplo, int n, int nrhs, complex *a, int lda, int *ipivot, complex *b, int ldb, int *info);

void csytrs_64(char uplo, long n, long nrhs, complex *a, long lda, long *ipivot, complex *b, long ldb, long *info);

PURPOSE

csytrs solves a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

ctbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE CTBCON( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NDIAG, LDA, INFO
 REAL RCOND
 REAL WORK2(*)

 SUBROUTINE CTBCON_64( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND,
*      WORK, WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, INFO
 REAL RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TBCON( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE TBCON_64( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL :: RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ctbcon(char norm, char uplo, char diag, int n, int ndiag, complex *a, int lda, float *rcond, int *info);

void ctbcon_64(char norm, char uplo, char diag, long n, long ndiag, complex *a, long lda, float *rcond, long *info);

PURPOSE

ctbcon estimates the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE CTBMV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER N, NDIAG, LDA, INCY

SUBROUTINE CTBMV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, NDIAG, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TBMV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INCY

SUBROUTINE TBMV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ctbmv(char uplo, char transa, char diag, int n, int ndiag, complex *a, int lda, complex *y, int incy);

void ctbmv_64(char uplo, char transa, char diag, long n, long ndiag, complex *a, long lda, complex *y, long incy);

PURPOSE

ctbmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The

top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE CTBRFS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CTBRFS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TBRFS( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TBRFS_64( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A,
*       [LDA], B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ctbrfs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, complex *a, int lda, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void ctbrfs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, complex *a, long lda, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

ctbrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix.

The solution matrix X must be computed by CTBTRS or some other means before entering this routine. CTBRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(2*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctbsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE CTBSV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER N, NDIAG, LDA, INCY

SUBROUTINE CTBSV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, NDIAG, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TBSV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INCY

SUBROUTINE TBSV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ctbsv(char uplo, char transa, char diag, int n, int ndiag, complex *a, int lda, complex *y, int incy);

void ctbsv_64(char uplo, char transa, char diag, long n, long ndiag, complex *a, long lda, complex *y, long incy);

PURPOSE

ctbsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper

triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctbtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE CTBTRS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

 SUBROUTINE CTBTRS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
 SUBROUTINE TBTRS( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

 SUBROUTINE TBTRS_64( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ctbtrs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void ctbtrs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

ctbtrs solves a triangular system of the form

where A is a triangular band matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of A. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A

are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.
```

NAME

ctgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B)

SYNOPSIS

```
 SUBROUTINE CTGEVC( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 REAL RWORK(*)

 SUBROUTINE CTGEVC_64( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEVC( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE TGEVC_64( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
```

```
INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void ctgevc(char side, char howmny, logical *select, int n, complex *a, int lda, complex *b, int ldb, complex *vl, int ldvl, complex *vr, int ldvr, int mm, int *m, int *info);

void ctgevc_64(char side, char howmny, logical *select, long n, complex *a, long lda, complex *b, long ldb, complex *vl, long ldvl, complex *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

ctgevc computes some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B).

The right generalized eigenvector x and the left generalized eigenvector y of (A,B) corresponding to a generalized eigenvalue w are defined by:

```
(A - wB) * x = 0  and  y**H * (A - wB) = 0
```

where y**H denotes the conjugate tranpose of y.

If an eigenvalue w is determined by zero diagonal elements of both A and B, a unit vector is returned as the corresponding eigenvector.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of (A,B), or the products Z*X and/or Q*Y, where Z and Q are input unitary matrices. If (A,B) was obtained from the generalized Schur factorization of an original pair of matrices

```
(A0,B0) = (Q*A*Z**H,Q*B*Z**H),
```

then Z*X and Q*Y are the matrices of right or left eigenvectors of A.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R': compute right eigenvectors only;

  = 'L': compute left eigenvectors only;

  = 'B': compute both right and left eigenvectors.
  ```

- **HOWMNY (input)**

```
 = 'A': compute all right and/or left eigenvectors;

 = 'B': compute all right and/or left eigenvectors, and
backtransform them using the input matrices supplied
in VR and/or VL;
 = 'S': compute selected right and/or left eigenvectors,
specified by the logical array SELECT.
```

- **SELECT (input)**
  If HOWMNY ='S', SELECT specifies the eigenvectors to be computed. If HOWMNY ='A' or 'B', SELECT is not referenced. To select the eigenvector corresponding to the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input)**
  The upper triangular matrix A.
- **LDA (input)**
  The leading dimension of array A. LDA >= max(1,N).
- **B (input)**
  The upper triangular matrix B. B must have real diagonal elements.
- **LDB (input)**
  The leading dimension of array B. LDB >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the unitary matrix Q of left Schur vectors returned by CHGEQZ). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of (A,B); if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the unitary matrix Z of right Schur vectors returned by CHGEQZ). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of (A,B); if HOWMNY = 'B', the matrix Z*X; if HOWMNY = 'S', the right eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected eigenvector occupies one column.
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

ctgexc - reorder the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST

SYNOPSIS

```
 SUBROUTINE CTGEXC( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z, LDZ,
*       IFST, ILST, INFO)
 COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL WANTQ, WANTZ

 SUBROUTINE CTGEXC_64( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z,
*       LDZ, IFST, ILST, INFO)
 COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER*8 N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL*8 WANTQ, WANTZ
```

F95 INTERFACE

```
 SUBROUTINE TGEXC( WANTQ, WANTZ, [N], A, [LDA], B, [LDB], Q, [LDQ],
*       Z, [LDZ], IFST, ILST, [INFO])
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL :: WANTQ, WANTZ

 SUBROUTINE TGEXC_64( WANTQ, WANTZ, [N], A, [LDA], B, [LDB], Q, [LDQ],
*       Z, [LDZ], IFST, ILST, [INFO])
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER(8) :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL(8) :: WANTQ, WANTZ
```

C INTERFACE

#include <sunperf.h>

void ctgexc(logical wantq, logical wantz, int n, complex *a, int lda, complex *b, int ldb, complex *q, int ldq, complex *z, int ldz, int *ifst, int *ilst, int *info);

void ctgexc_64(logical wantq, logical wantz, long n, complex *a, long lda, complex *b, long ldb, complex *q, long ldq, complex *z, long ldz, long *ifst, long *ilst, long *info);

PURPOSE

ctgexc reorders the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST.

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Optionally, the matrices Q and Z of generalized Schur vectors are updated.

```
Q(in) * A(in) * Z(in)' = Q(out) * A(out) * Z(out)'
Q(in) * B(in) * Z(in)' = Q(out) * B(out) * Z(out)'
```

ARGUMENTS

- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper triangular matrix A in the pair (A, B). On exit, the updated matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the upper triangular matrix B in the pair (A, B). On exit, the updated matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., the unitary matrix Q. On exit, the updated matrix Q. If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; If WANTQ = .TRUE., LDQ >= N.
- **Z (input/output)**

On entry, if WANTZ = .TRUE., the unitary matrix Z. On exit, the updated matrix Z. If WANTZ = .FALSE., Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1; If WANTZ = .TRUE., LDZ > = N.
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of (A, B). The block with row index IFST is moved to row ILST, by a sequence of swapping between adjacent blocks.
- **ILST (input/output)**
  See the description of IFST.
- **INFO (output)**

```
 =0:  Successful exit.

 <0:  if INFO  = -i, the i-th argument had an illegal value.

 =1:  The transformed matrix pair (A, B) would be too far
from generalized Schur form; the problem is ill-
conditioned. (A, B) may have been partially reordered,
and ILST points to the first row of the current
position of the block being moved.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

```
    UMINF - 94.04, Department of Computing Science, Umea University,
    S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
    To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ctgsen - reorder the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B)

SYNOPSIS

```
 SUBROUTINE CTGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL WANTQ, WANTZ
 LOGICAL SELECT(*)
 REAL PL, PR
 REAL DIF(*)

 SUBROUTINE CTGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 WANTQ, WANTZ
 LOGICAL*8 SELECT(*)
 REAL PL, PR
 REAL DIF(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSEN( IJOB, WANTQ, WANTZ, SELECT, [N], A, [LDA], B, [LDB],
*       ALPHA, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
```

```
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL :: WANTQ, WANTZ
  LOGICAL, DIMENSION(:) :: SELECT
  REAL :: PL, PR
  REAL, DIMENSION(:) :: DIF

  SUBROUTINE TGSEN_64( IJOB, WANTQ, WANTZ, SELECT, [N], A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
  COMPLEX, DIMENSION(:) :: ALPHA, BETA, WORK
  COMPLEX, DIMENSION(:,:) :: A, B, Q, Z
  INTEGER(8) :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8) :: WANTQ, WANTZ
  LOGICAL(8), DIMENSION(:) :: SELECT
  REAL :: PL, PR
  REAL, DIMENSION(:) :: DIF
```

C INTERFACE

#include <sunperf.h>

void ctgsen(int ijob, logical wantq, logical wantz, logical *select, int n, complex *a, int lda, complex *b, int ldb, complex *alpha, complex *beta, complex *q, int ldq, complex *z, int ldz, int *m, float *pl, float *pr, float *dif, int *info);

void ctgsen_64(long ijob, logical wantq, logical wantz, logical *select, long n, complex *a, long lda, complex *b, long ldb, complex *alpha, complex *beta, complex *q, long ldq, complex *z, long ldz, long *m, float *pl, float *pr, float *dif, long *info);

PURPOSE

ctgsen reorders the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B). The leading columns of Q and Z form unitary bases of the corresponding left and right eigenspaces (deflating subspaces). (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

CTGSEN also computes the generalized eigenvalues

```
          w(j)= ALPHA(j) / BETA(j)
```

of the reordered matrix pair (A, B).

Optionally, the routine computes estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[(A11,B11), (A22,B22)] and Difl[(A11,B11), (A22,B22)], i.e. the `separation(s)` between the matrix pairs (A11, B11) and (A22,B22) that correspond to the selected cluster and the eigenvalues outside the cluster, resp., and norms of ``projections'' onto left and right eigenspaces w.r.t. the selected cluster in the (1,1)-block.

ARGUMENTS

- **IJOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (PL and PR) or the deflating subspaces (Difu and Difl):

  ```
  =0: Only reorder w.r.t. SELECT. No extras.

  =1: Reciprocal of norms of "projections" onto left and right
eigenspaces w.r.t. the selected cluster (PL and PR).
  =2: Upper bounds on Difu and Difl. F-norm-based estimate
  ```

  (DIF(1:2)).

  ```
  =3: Estimate of Difu and Difl. 1-norm-based estimate
  ```

  (DIF(1:2)). About 5 times as expensive as IJOB = 2. =4: Compute PL, PR and DIF (i.e. 0, 1 and 2 above): Economic version to get it all. =5: Compute PL, PR and DIF (i.e. 0, 1 and 3 above)
- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select an eigenvalue w(j), `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper triangular matrix A, in generalized Schur canonical form. On exit, A is overwritten by the reordered matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the upper triangular matrix B, in generalized Schur canonical form. On exit, B is overwritten by the reordered matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ALPHA (output)**
  The diagonal elements of A and B, respectively, when the pair (A,B) has been reduced to generalized Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues.
- **BETA (output)**
  See the description of ALPHA.
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., Q is an N-by-N matrix. On exit, Q has been postmultiplied by the left unitary transformation matrix which reorder (A, B); The leading M columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1. If WANTQ = .TRUE., LDQ >= N.

- **Z (input/output)**
  On entry, if WANTZ = .TRUE., Z is an N-by-N matrix. On exit, Z has been postmultiplied by the left unitary transformation matrix which reorder (A, B); The leading M columns of Z form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1. If WANTZ = .TRUE., LDZ >= N.
- **M (output)**
  The dimension of the specified pair of left and right eigenspaces, (deflating subspaces) 0 <= M <= N.
- **PL (output)**
  IF IJOB = 1, 4, or 5, PL, PR are lower bounds on the reciprocal of the norm of ``projections'' onto left and right eigenspace with respect to the selected cluster.

  0 < PL, PR <= 1. If M = 0 or M = N, PL = PR = 1. If IJOB = 0, 2, or 3 PL, PR are not referenced.
- **PR (output)**
  See the description of PL.
- **DIF (output)**
  If IJOB >= 2, DIF(1:2) store the estimates of Difu and Difl.

  If IJOB = 2 or 4, DIF(1:2) are F-norm-based upper bounds on

  Difu and Difl. If IJOB = 3 or 5, DIF(1:2) are 1-norm-based estimates of Difu and Difl, computed using reversed communication with CLACON. If M = 0 or N, DIF(1:2) = F-norm([A, B]). If IJOB = 0 or 1, DIF is not referenced.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1 If IJOB = 1, 2 or 4, LWORK >= 2*M*(N-M) If IJOB = 3 or 5, LWORK >= 4*M*(N-M)

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced. Otherwise, on exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 1. If IJOB = 1, 2 or 4, LIWORK >= N+2; If IJOB = 3 or 5, LIWORK >= MAX(N+2, 2*M*(N-M));

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 =0: Successful exit.

 <0: If INFO  = -i, the i-th argument had an illegal value.

 =1: Reordering of (A, B) failed because the transformed
matrix pair (A, B) would be too far from generalized
Schur form; the problem is very ill-conditioned.
(A, B) may have been partially reordered.
If requested, 0 is returned in DIF(*), PL and PR.
```

FURTHER DETAILS

CTGSEN first collects the selected eigenvalues by computing unitary U and W that move them to the top left corner of (A, B). In other words, the selected eigenvalues are the eigenvalues of (A11, B11) in

```
U'*(A, B)*W  = (A11 A12) (B11 B12) n1
               ( 0  A22),( 0  B22) n2
                 n1  n2    n1  n2
```

where N = n1+n2 and U' means the conjugate transpose of U. The first n1 columns of U and W span the specified pair of left and right eigenspaces (deflating subspaces) of (A, B).

If (A, B) has been obtained from the generalized real Schur decomposition of a matrix pair (C, D) = Q*(A, B)*Z', then the reordered generalized Schur form of (C, D) is given by

```
(C, D)  = (Q*U)*(U'*(A, B)*W)*(Z*W)',
```

and the first n1 columns of Q*U and Z*W span the corresponding deflating subspaces of (C, D) (Q and Z store Q*U and Z*W, resp.).

Note that if the selected eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

The reciprocal condition numbers of the left and right eigenspaces spanned by the first n1 columns of U and W (or Q*U and Z*W) may be returned in DIF(1:2), corresponding to Difu and Difl, resp.

The Difu and Difl are defined as:

ifu[(A11, B11), (A22, B22)] = sigma-min(Zu)

and ifl[(A11, B11), (A22, B22)] = Difu[(A22, B22), (A11, B11)],

where sigma-min(Zu) is the smallest singular value of the (2*n1*n2)-by-(2*n1*n2) matrix

u = [kron(In2, A11) -kron(A22', In1)]

```
    [ kron(In2, B11)  -kron(B22', In1) ].
```

Here, Inx is the identity matrix of size nx and A22' is the transpose of A22. kron(X, Y) is the Kronecker product between the matrices X and Y.

When DIF(2) is small, small changes in (A, B) can cause large changes in the deflating subspace. An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is PS * norm((A, B)) / DIF(2),

where EPS is the machine precision.

The reciprocal norm of the projectors on the left and right eigenspaces associated with (A11, B11) may be returned in PL and PR. They are computed as follows. First we compute L and R so that P*(A, B)*Q is block diagonal, where

```
= ( I -L ) n1          Q  = ( I R ) n1
  ( 0  I ) n2    and         ( 0 I ) n2
    n1 n2                     n1 n2
```

and (L, R) is the solution to the generalized Sylvester equation 11*R - L*A22 = -A12 11*R - L*B22 = -B12

Then PL = (F-norm(L)**2+1)**(-1/2) and PR = (F-norm(R)**2+1)**(-1/2). An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

EPS * norm((A, B)) / PL.

There are also global error bounds which valid for perturbations up to a certain restriction: A lower bound (x) on the smallest F-norm(E,F) for which an eigenvalue of (A11, B11) may move and coalesce with an eigenvalue of (A22, B22) under perturbation (E,F), (i.e. (A + E, B + F), is

```
x  = min(Difu,Difl)/((1/(PL*PL)+1/(PR*PR))**(1/2)+2*max(1/PL,1/PR)).
```

An approximate bound on x can be computed from DIF(1:2), PL and PL.

If y = (F-norm(E,F) / x) < = 1, the angles between the perturbed (L', R') and unperturbed (L, R) left and right deflating subspaces associated with the selected cluster in the (1,1)-blocks can be bounded as

```
max-angle(L, L')  < = arctan( y * PL / (1 - y * (1 - PL * PL)**(1/2))
max-angle(R, R')  < = arctan( y * PR / (1 - y * (1 - PR * PR)**(1/2))
```

See LAPACK User's Guide section 4.11 or the following references for more information.

Note that if the default method for computing the Frobenius-norm- based estimate DIF is not wanted (see CLATDF), then the parameter IDIFJB (see below) should be changed from 3 to 4 (routine CLATDF (IJOB = 2 will be used)). See CTGSYL for more details.

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

References

```
 = = = = = = = = = =
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

```
    UMINF - 94.04, Department of Computing Science, Umea University,
    S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
    To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ctgsja - compute the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B

SYNOPSIS

```
 SUBROUTINE CTGSJA( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B, LDB,
*      TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, NCYCLE,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL TOLA, TOLB
 REAL ALPHA(*), BETA(*)

 SUBROUTINE CTGSJA_64( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B,
*      LDB, TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK,
*      NCYCLE, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER*8 M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL TOLA, TOLB
 REAL ALPHA(*), BETA(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSJA( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: ALPHA, BETA

 SUBROUTINE TGSJA_64( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
```

```
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER(8) :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: ALPHA, BETA
```

C INTERFACE

#include <sunperf.h>

void ctgsja(char jobu, char jobv, char jobq, int m, int p, int n, int k, int l, complex *a, int lda, complex *b, int ldb, float tola, float tolb, float *alpha, float *beta, complex *u, int ldu, complex *v, int ldv, complex *q, int ldq, int *ncycle, int *info);

void ctgsja_64(char jobu, char jobv, char jobq, long m, long p, long n, long k, long l, complex *a, long lda, complex *b, long ldb, float tola, float tolb, float *alpha, float *beta, complex *u, long ldu, complex *v, long ldv, complex *q, long ldq, long *ncycle, long *info);

PURPOSE

ctgsja computes the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B.

On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine CGGSVP from a general M-by-N matrix A and P-by-N matrix B:

```
             N-K-L  K    L

A =    K ( 0    A12  A13 ) if M-K-L >= 0;

       L ( 0     0   A23 )

   M-K-L ( 0     0    0  )

            N-K-L  K    L

A =  K ( 0    A12  A13 ) if M-K-L < 0;

   M-K ( 0     0   A23 )

            N-K-L  K    L

B =  L ( 0     0   B13 )

   P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal.

On exit,

```
       U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R ),
```

where U, V and Q are unitary matrices, Z' denotes the conjugate transpose of Z, R is a nonsingular upper triangular matrix, and D1 and D2 are ``diagonal" matrices, which are of the following structures:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
            M-K-L ( 0  0 )

                   K  L
       D2 = L   ( 0  S )
            P-L ( 0  0 )

               N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 ) K
            L (  0    0   R22 ) L
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),
  S = diag( BETA(K+1),  ... , BETA(K+L) ),
  C**2 + S**2 = I.
  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                 K M-K K+L-M
    D1 =   K ( I  0    0   )
         M-K ( 0  C    0   )
                   K M-K K+L-M
    D2 =   M-K ( 0  S    0   )
         K+L-M ( 0  0    I   )
```

```
             P-L ( 0  0    0   )

                  N-K-L  K   M-K  K+L-M

               M-K ( 0     0   R22  R23  )

             K+L-M ( 0     0    0   R33  )
```

where

C = diag(ALPHA(K+1), ... , ALPHA(M)),

S = diag(BETA(K+1), ... , BETA(M)),

C**2 + S**2 = I.

R = (R11 R12 R13) is stored in A(1:M, N-K-L+1:N) and R33 is stored (0 R22 R23)

in B(M-K+1:L,N+M-K-L+1:N) on exit.

The computation of the unitary transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U1, V1, or Q1.

CTGSJA essentially uses a variant of Kogbetliantz algorithm to reduce min(L,M-K)-by-L triangular (or trapezoidal) matrix A23 and L-by-L matrix B13 to the form:

```
U1'*A13*Q1 = C1*R1; V1'*B13*Q1 = S1*R1,
```

where U1, V1 and Q1 are unitary matrix, and Z' is the conjugate transpose of Z. C1 and S1 are diagonal matrices satisfying

```
C1**2 + S1**2 = I,
```

and R1 is an L-by-L nonsingular upper triangular matrix.

ARGUMENTS

- **JOBU (input)**

```
 = 'U':  U must contain a unitary matrix U1 on entry, and
the product U1*U is returned;
 = 'I':  U is initialized to the unit matrix, and the
unitary matrix U is returned;
 = 'N':  U is not computed.
```

- **JOBV (input)**

```
 = 'V':  V must contain a unitary matrix V1 on entry, and
the product V1*V is returned;
 = 'I':  V is initialized to the unit matrix, and the
unitary matrix V is returned;
 = 'N':  V is not computed.
```

- **JOBQ (input)**

```
 = 'Q':  Q must contain a unitary matrix Q1 on entry, and
the product Q1*Q is returned;
 = 'I':  Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'N':  Q is not computed.
```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **K (input)**
  K and L specify the subblocks in the input matrices A and B:

  A23 = A(K+1:MIN(K+L,M),N-L+1:N) and B13 = B(1:L,,N-L+1:N) of A and B, whose GSVD is going to be computed by CTGSJA. See the Further Details section below.
- **L (input)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A(N-K+1:N,1:MIN(K+L,M)) contains the triangular matrix R or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, if necessary, B(M-K+1:L,N+M-K-L+1:N) contains a part of R. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the convergence criteria for the Jacobi- Kogbetliantz iteration procedure. Generally, they are the same as used in the preprocessing step, say TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS.
- **TOLB (input)**
  See the description of TOLA.
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L >= 0, ALPHA(K+1:K+L) = diag(C),

  BETA(K+1:K+L) = diag(S), or if M-K-L < 0, ALPHA(K+1:M) = C, ALPHA(M+1:K+L) = 0

  BETA(K+1:M) = S, BETA(M+1:K+L) = 1. Furthermore, if K+L < N, ALPHA(K+L+1:N) = 0

  BETA(K+L+1:N) = 0.
- **BETA (output)**
  See the description of ALPHA.
- **U (input/output)**
  On entry, if JOBU = 'U', U must contain a matrix U1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBU = 'I', U contains the unitary matrix U; if JOBU = 'U', U contains the product U1*U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (input/output)**

On entry, if JOBV = 'V', V must contain a matrix V1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBV = 'I', V contains the unitary matrix V; if JOBV = 'V', V contains the product V1*V. If JOBV = 'N', V is not referenced.

- **LDV (input)**
  The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.
- **Q (input/output)**
  On entry, if JOBQ = 'Q', Q must contain a matrix Q1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBQ = 'I', Q contains the unitary matrix Q; if JOBQ = 'Q', Q contains the product Q1*Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **NCYCLE (output)**
  The number of cycles required for convergence.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.

= 1:  the procedure does not converge after MAXIT cycles.
```

NAME

ctgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B)

SYNOPSIS

```
 SUBROUTINE CTGSNA( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL, LDVL,
*      VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 REAL S(*), DIF(*)

 SUBROUTINE CTGSNA_64( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 REAL S(*), DIF(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSNA( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, DIF
```

```
 SUBROUTINE TGSNA_64( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, DIF
```

C INTERFACE

#include <sunperf.h>

void ctgsna(char job, char howmnt, logical *select, int n, complex *a, int lda, complex *b, int ldb, complex *vl, int ldvl, complex *vr, int ldvr, float *s, float *dif, int mm, int *m, int *info);

void ctgsna_64(char job, char howmnt, logical *select, long n, complex *a, long lda, complex *b, long ldb, complex *vl, long ldvl, complex *vr, long ldvr, float *s, float *dif, long mm, long *m, long *info);

PURPOSE

ctgsna estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B).

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (DIF):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (DIF);

  = 'B': for both eigenvalues and eigenvectors (S and DIF).
  ```
- **HOWMNT (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNT = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the corresponding j-th eigenvalue and/or eigenvector, `SELECT(j)` must be set to .TRUE.. If HOWMNT = 'A', SELECT is not referenced.
- **N (input)**

The order of the square matrix pair (A, B). N > = 0.

- **A (input)**
  The upper triangular matrix A in the pair (A,B).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input)**
  The upper triangular matrix B in the pair (A, B).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by CTGEVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and If JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by CTGEVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; If JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If JOB = 'V', S is not referenced.
- **DIF (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute DIF(j), `DIF(j)` is set to 0; this can only occur when the true value would be very small anyway. For each eigenvalue/vector specified by SELECT, DIF stores a Frobenius norm-based estimate of Difl. If JOB = 'E', DIF is not referenced.
- **MM (input)**
  The number of elements in the arrays S and DIF. MM > = M.
- **M (output)**
  The number of elements of the arrays S and DIF used to store the specified condition numbers; for each selected eigenvalue one element is used. If HOWMNT = 'A', M is set to N.
- **WORK (workspace)**
  If JOB = 'E', WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. If JOB = 'V' or 'B', LWORK > = 2*N*N.
- **IWORK (workspace)**
  `dimension(N+2)` If JOB = 'E', IWORK is not referenced.
- **INFO (output)**

```
= 0: Successful exit

< 0: If INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of the i-th generalized eigenvalue w = (a, b) is defined as

```
S(I)  = (|v'Au|**2 + |v'Bu|**2)**(1/2) / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of (A, B) corresponding to w; |z| denotes the absolute value of the complex number, and `norm(u)` denotes the 2-norm of the vector u. The pair (a, b) corresponds to an eigenvalue w = a/b (= v'Au/v'Bu) of the matrix pair (A, B). If both a and b equal zero, then (A,B) is singular and `S(I)` = -1 is returned.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

```
chord(w, lambda)  < =   EPS * norm(A, B) / S(I),
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u and left eigenvector v corresponding to the generalized eigenvalue w is defined as follows. Suppose

```
(A, B)  = ( a   *  ) ( b  *  )  1

          ( 0  A22 ),( 0 B22 )  n-1

            1  n-1     1 n-1
```

Then the reciprocal condition number `DIF(I)` is

```
Difl[(a, b), (A22, B22)]   = sigma-min( Zl )
```

where sigma-min(Zl) denotes the smallest singular value of

```
Zl  = [ kron(a, In-1) -kron(1, A22) ]

      [ kron(b, In-1) -kron(1, B22) ].
```

Here In-1 is the identity matrix of size n-1 and X' is the conjugate transpose of X. kron(X, Y) is the Kronecker product between the matrices X and Y.

We approximate the smallest singular value of Zl with an upper bound. This is done by CLATDF.

An approximate error bound for a computed eigenvector `VL(i)` or `VR(i)` is given by

```
EPS * norm(A, B) / DIF(i).
```

See ref. [2-3] for more details and further references.

Based on contributions by


```
Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.
```


References

= = = = = = = = = =

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

UMINF - 94.04, Department of Computing Science, Umea University,
S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
To appear in Numerical Algorithms, 1996.

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75.

To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ctgsyl - solve the generalized Sylvester equation

SYNOPSIS

```
 SUBROUTINE CTGSYL( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
 INTEGER IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER IWORK(*)
 REAL SCALE, DIF

 SUBROUTINE CTGSYL_64( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
 INTEGER*8 IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL SCALE, DIF
```

F95 INTERFACE

```
 SUBROUTINE TGSYL( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C, [LDC],
*       D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: TRANS
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, C, D, E, F
 INTEGER :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: SCALE, DIF

 SUBROUTINE TGSYL_64( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C,
*       [LDC], D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK],
*       [IWORK], [INFO])
```

```
CHARACTER(LEN=1) :: TRANS
COMPLEX, DIMENSION(:) :: WORK
COMPLEX, DIMENSION(:,:) :: A, B, C, D, E, F
INTEGER(8) :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL :: SCALE, DIF
```

C INTERFACE

#include <sunperf.h>

void ctgsyl(char trans, int ijob, int m, int n, complex *a, int lda, complex *b, int ldb, complex *c, int ldc, complex *d, int ldd, complex *e, int lde, complex *f, int ldf, float *scale, float *dif, int *info);

void ctgsyl_64(char trans, long ijob, long m, long n, complex *a, long lda, complex *b, long ldb, complex *c, long ldc, complex *d, long ldd, complex *e, long lde, complex *f, long ldf, float *scale, float *dif, long *info);

PURPOSE

ctgsyl solves the generalized Sylvester equation:

```
A * R - L * B = scale * C            (1)

D * R - L * E = scale * F
```

where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively, with complex entries. A, B, D and E are upper triangular (i.e., (A,D) and (B,E) in generalized Schur form).

The solution (R, L) overwrites (C, F). 0 <= SCALE <= 1

is an output scaling factor chosen to avoid overflow.

In matrix notation (1) is equivalent to solve Zx = scale*b, where Z is defined as

```
Z = [ kron(In, A)  -kron(B', Im) ]        (2)

    [ kron(In, D)  -kron(E', Im) ],
```

Here Ix is the identity matrix of size x and X' is the conjugate transpose of X. Kron(X, Y) is the Kronecker product between the matrices X and Y.

If TRANS = 'C', y in the conjugate transposed system Z'*y = scale*b is solved for, which is equivalent to solve for R and L in

```
A' * R + D' * L = scale * C           (3)

R * B' + L * E' = scale * -F
```

This case (TRANS = 'C') is used to compute an one-norm-based estimate of Dif[(A,D), (B,E)], the separation between the matrix pairs (A,D) and (B,E), using CLACON.

If IJOB >= 1, CTGSYL computes a Frobenius norm-based estimate of Dif[(A,D),(B,E)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z.

This is a level-3 BLAS algorithm.

ARGUMENTS

- **TRANS (input)**

```
= 'N': solve the generalized sylvester equation (1).

= 'C': solve the "conjugate transposed" system (3).
```

- **IJOB (input)**
  Specifies what kind of functionality to be performed. =0: solve (1) only.

```
=1: The functionality of 0 and 3.

=2: The functionality of 0 and 4.

=3: Only an estimate of Dif[(A,D), (B,E)] is computed.
(look ahead strategy is used).
=4: Only an estimate of Dif[(A,D), (B,E)] is computed.
(CGECON on sub-systems is used).
Not referenced if TRANS  = 'C'.
```

- **M (input)**
  The order of the matrices A and D, and the row dimension of the matrices C, F, R and L.
- **N (input)**
  The order of the matrices B and E, and the column dimension of the matrices C, F, R and L.
- **A (input)**
  The upper triangular matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1, M).
- **B (input)**
  The upper triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1, N).
- **C (input/output)**
  On entry, C contains the right-hand-side of the first matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, C has been overwritten by the solution R. If IJOB = 3 or 4 and TRANS = 'N', C holds R, the solution achieved during the computation of the Dif-estimate.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1, M).
- **D (input)**
  The upper triangular matrix D.
- **LDD (input)**
  The leading dimension of the array D. LDD >= max(1, M).
- **E (input)**
  The upper triangular matrix E.
- **LDE (input)**
  The leading dimension of the array E. LDE >= max(1, N).
- **F (input/output)**

On entry, F contains the right-hand-side of the second matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, F has been overwritten by the solution L. If IJOB = 3 or 4 and TRANS = 'N', F holds L, the solution achieved during the computation of the Dif-estimate.

- **LDF (input)**
  The leading dimension of the array F. LDF > = max(1, M).
- **SCALE (output)**
  On exit DIF is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. DIF is an upper bound of Dif[(A,D), (B,E)] = sigma-min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'C', DIF is not referenced.
- **DIF (output)**
  On exit DIF is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. DIF is an upper bound of Dif[(A,D), (B,E)] = sigma-min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'C', DIF is not referenced.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO =0 then WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. If IJOB = 1 or 2 and TRANS = 'N', LWORK > = 2*M*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced.
- **INFO (output)**

```
 =0: successful exit

 <0: If INFO  = -i, the i-th argument had an illegal value.

 >0: (A, D) and (B, E) have common or very close
eigenvalues.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

[2] B. Kagstrom, A Perturbation Analysis of the Generalized Sylvester Equation (AR - LB, DR - LE) = (C, F), SIAM J. Matrix Anal. Appl., 15(4):1045-1060, 1994.

[3] B. Kagstrom and L. Westin, Generalized Schur Methods with Condition Estimators for Solving the Generalized Sylvester Equation, IEEE Transactions on Automatic Control, Vol. 34, No. 7, July 1989, pp 745-751.

NAME

ctpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
  SUBROUTINE CTPCON( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2, INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  COMPLEX A(*), WORK(*)
  INTEGER N, INFO
  REAL RCOND
  REAL WORK2(*)

  SUBROUTINE CTPCON_64( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2,
 *      INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  COMPLEX A(*), WORK(*)
  INTEGER*8 N, INFO
  REAL RCOND
  REAL WORK2(*)
```

F95 INTERFACE

```
  SUBROUTINE TPCON( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
 *       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  COMPLEX, DIMENSION(:) :: A, WORK
  INTEGER :: N, INFO
  REAL :: RCOND
  REAL, DIMENSION(:) :: WORK2

  SUBROUTINE TPCON_64( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
 *       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  COMPLEX, DIMENSION(:) :: A, WORK
  INTEGER(8) :: N, INFO
  REAL :: RCOND
  REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ctpcon(char norm, char uplo, char diag, int n, complex *a, float *rcond, int *info);

void ctpcon_64(char norm, char uplo, char diag, long n, complex *a, float *rcond, long *info);

PURPOSE

ctpcon estimates the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

ctpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE CTPMV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), Y(*)
INTEGER N, INCY

SUBROUTINE CTPMV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE TPMV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A, Y
INTEGER :: N, INCY

SUBROUTINE TPMV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A, Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void ctpmv(char uplo, char transa, char diag, int n, complex *a, complex *y, int incy);

void ctpmv_64(char uplo, char transa, char diag, long n, complex *a, complex *y, long incy);

PURPOSE

ctpmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE CTPRFS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CTPRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TPRFS( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB], X,
*        [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: A, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TPRFS_64( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: A, WORK
 COMPLEX, DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ctprfs(char uplo, char transa, char diag, int n, int nrhs, complex *a, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void ctprfs_64(char uplo, char transa, char diag, long n, long nrhs, complex *a, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

ctprfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix.

The solution matrix X must be computed by CTPTRS or some other means before entering this routine. CTPRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctpsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE CTPSV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), Y(*)
INTEGER N, INCY

SUBROUTINE CTPSV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE TPSV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A, Y
INTEGER :: N, INCY

SUBROUTINE TPSV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A, Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void ctpsv(char uplo, char transa, char diag, int n, complex *a, complex *y, int incy);

void ctpsv_64(char uplo, char transa, char diag, long n, complex *a, complex *y, long incy);

PURPOSE

ctpsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctptri - compute the inverse of a complex upper or lower triangular matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE CTPTRI( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE CTPTRI_64( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE TPTRI( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE TPTRI_64( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void ctptri(char uplo, char diag, int n, complex *a, int *info);

void ctptri_64(char uplo, char diag, long n, complex *a, long *info);

PURPOSE

ctptri computes the inverse of a complex upper or lower triangular matrix A stored in packed format.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangular matrix A, stored columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = A(i,j) for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*((2*n-j)/2) = A(i,j) for j < =i < =n. See below for further details. On exit, the (triangular) inverse of the original matrix, in the same packed storage format.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

FURTHER DETAILS

A triangular matrix A can be transferred to packed storage using one of the following program segments:

UPLO = 'U': UPLO = 'L':

```
JC  = 1                          JC  = 1

DO 2 J  = 1, N                   DO 2 J  = 1, N
```

```
     DO 1 I  = 1, J
        A(JC+I-1)  = A(I,J)
1    CONTINUE
     JC  = JC + J
2 CONTINUE
```

```
      DO 1 I  = J, N
        A(JC+I-J)  = A(I,J)
1    CONTINUE
      JC  = JC + N - J + 1
2 CONTINUE
```

NAME

ctptrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
SUBROUTINE CTPTRS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE CTPTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE TPTRS( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE TPTRS_64( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: A
COMPLEX, DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ctptrs(char uplo, char transa, char diag, int n, int nrhs, complex *a, complex *b, int ldb, int *info);

void ctptrs_64(char uplo, char transa, char diag, long n, long nrhs, complex *a, complex *b, long ldb, long *info);

PURPOSE

ctptrs solves a triangular system of the form

where A is a triangular matrix of order N stored in packed format, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose)
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

 < 0: if INFO = -i, the i-th argument had an illegal value

 > 0: if INFO = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.

NAME

ctrans - transpose and scale source matrix

SYNOPSIS

```
SUBROUTINE CTRANS( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
COMPLEX SCALE
COMPLEX SOURCE(*), DEST(*)
INTEGER M, N

SUBROUTINE CTRANS_64( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
COMPLEX SCALE
COMPLEX SOURCE(*), DEST(*)
INTEGER*8 M, N
```

F95 INTERFACE

```
SUBROUTINE TRANS( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
COMPLEX :: SCALE
COMPLEX, DIMENSION(:) :: SOURCE, DEST
INTEGER :: M, N

SUBROUTINE TRANS_64( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
COMPLEX :: SCALE
COMPLEX, DIMENSION(:) :: SOURCE, DEST
INTEGER(8) :: M, N
```

C INTERFACE

#include <sunperf.h>

void ctrans(char place, complex scale, complex *source, int m, int n, complex *dest);

void ctrans_64(char place, complex scale, complex *source, long m, long n, complex *dest);

PURPOSE

ctrans scales and transposes the source matrix. The N2 x N1 result is written into SOURCE when PLACE = 'I' or 'i', and DEST when PLACE = 'O' or 'o'.

```
PLACE = 'I' or 'i': SOURCE = SCALE * SOURCE'

PLACE = 'O' or 'o': DEST = SCALE * SOURCE'
```

ARGUMENTS

- **PLACE (input)**
  Type of transpose. 'I' or 'i' for in-place, 'O' or 'o' for out-of-place. 'I' is default.
- **SCALE (input)**
  Scale factor on the SOURCE matrix.
- **SOURCE (input/output)**
  on input. Array of (N, M) on output if in-place transpose.
- **M (input)**
  Number of rows in the SOURCE matrix on input.
- **N (input)**
  Number of columns in the SOURCE matrix on input.
- **DEST (output)**
  Scaled and transposed SOURCE matrix if out-of-place transpose. Not referenced if in-place transpose.

NAME

ctrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE CTRCON( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 REAL RCOND
 REAL WORK2(*)

 SUBROUTINE CTRCON_64( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 REAL RCOND
 REAL WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TRCON( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK2

 SUBROUTINE TRCON_64( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL :: RCOND
REAL, DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ctrcon(char norm, char uplo, char diag, int n, complex *a, int lda, float *rcond, int *info);

void ctrcon_64(char norm, char uplo, char diag, long n, complex *a, long lda, float *rcond, long *info);

PURPOSE

ctrcon estimates the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctrevc - compute some or all of the right and/or left eigenvectors of a complex upper triangular matrix T

SYNOPSIS

```
 SUBROUTINE CTREVC( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 REAL RWORK(*)

 SUBROUTINE CTREVC_64( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 REAL RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREVC( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: T, VL, VR
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: RWORK

 SUBROUTINE TREVC_64( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX, DIMENSION(:) :: WORK
```

```
COMPLEX, DIMENSION(:,:) :: T, VL, VR
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL, DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void ctrevc(char side, char howmny, logical *select, int n, complex *t, int ldt, complex *vl, int ldvl, complex *vr, int ldvr, int mm, int *m, int *info);

void ctrevc_64(char side, char howmny, logical *select, long n, complex *t, long ldt, complex *vl, long ldvl, complex *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

ctrevc computes some or all of the right and/or left eigenvectors of a complex upper triangular matrix T.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:

```
T*x = w*x,     y'*T = w*y'
```

where y' denotes the conjugate transpose of the vector y.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of T, or the products Q*X and/or Q*Y, where Q is an input unitary

matrix. If T was obtained from the Schur factorization of an original matrix A = Q*T*Q', then Q*X and Q*Y are the matrices of right or left eigenvectors of A.

ARGUMENTS

- **SIDE (input)**

```
= 'R':  compute right eigenvectors only;

= 'L':  compute left eigenvectors only;

= 'B':  compute both right and left eigenvectors.
```

- **HOWMNY (input)**

```
= 'A':  compute all right and/or left eigenvectors;

 = 'B':  compute all right and/or left eigenvectors,
and backtransform them using the input matrices
supplied in VR and/or VL;
 = 'S':  compute selected right and/or left eigenvectors,
```

specified by the logical array SELECT.

- **SELECT (input/output)**
  If HOWMNY = 'S', SELECT specifies the eigenvectors to be computed. If HOWMNY = 'A' or 'B', SELECT is not referenced. To select the eigenvector corresponding to the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  The upper triangular matrix T. T is modified, but restored on exit.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the unitary matrix Q of Schur vectors returned by CHSEQR). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of T; VL is lower triangular. The i-th column `VL(i)` of VL is the eigenvector corresponding to T(i,i). if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of T specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the unitary matrix Q of Schur vectors returned by CHSEQR). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of T; VR is upper triangular. The i-th column `VR(i)` of VR is the eigenvector corresponding to T(i,i). if HOWMNY = 'B', the matrix Q*X; if HOWMNY = 'S', the right eigenvectors of T specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected eigenvector occupies one column.
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The algorithm used in this program is basically backward (forward) substitution, with scaling to make the the code robust against possible overflow.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x| + |y|.

NAME

ctrexc - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST

SYNOPSIS

```
SUBROUTINE CTREXC( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, INFO)
CHARACTER * 1 COMPQ
COMPLEX T(LDT,*), Q(LDQ,*)
INTEGER N, LDT, LDQ, IFST, ILST, INFO

SUBROUTINE CTREXC_64( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, INFO)
CHARACTER * 1 COMPQ
COMPLEX T(LDT,*), Q(LDQ,*)
INTEGER*8 N, LDT, LDQ, IFST, ILST, INFO
```

F95 INTERFACE

```
SUBROUTINE TREXC( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST, [INFO])
CHARACTER(LEN=1) :: COMPQ
COMPLEX, DIMENSION(:,:) :: T, Q
INTEGER :: N, LDT, LDQ, IFST, ILST, INFO

SUBROUTINE TREXC_64( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST,
*       [INFO])
CHARACTER(LEN=1) :: COMPQ
COMPLEX, DIMENSION(:,:) :: T, Q
INTEGER(8) :: N, LDT, LDQ, IFST, ILST, INFO
```

C INTERFACE

#include <sunperf.h>

void ctrexc(char compq, int n, complex *t, int ldt, complex *q, int ldq, int ifst, int ilst, int *info);

void ctrexc_64(char compq, long n, complex *t, long ldt, complex *q, long ldq, long ifst, long ilst, long *info);

PURPOSE

ctrexc reorders the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST.

The Schur form T is reordered by a unitary similarity transformation Z**H*T*Z, and optionally the matrix Q of Schur vectors is updated by postmultplying it with Z.

ARGUMENTS

- **COMPQ (input)**

```
= 'V':  update the matrix Q of Schur vectors;

= 'N':  do not update Q.
```

- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input/output)**
  On entry, the upper triangular matrix T. On exit, the reordered upper triangular matrix.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the unitary transformation matrix Z which reorders T. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **IFST (input)**
  Specify the reordering of the diagonal elements of T: The element with row index IFST is moved to row ILST by a sequence of transpositions between adjacent elements. 1 < = IFST < = N; 1 < = ILST < = N.
- **ILST (input)**
  See the description of IFST.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

SYNOPSIS

```
 SUBROUTINE CTRMM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER M, N, LDA, LDB

 SUBROUTINE CTRMM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 M, N, LDA, LDB
```

F95 INTERFACE

```
 SUBROUTINE TRMM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, LDA, LDB

 SUBROUTINE TRMM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, LDA, LDB
```

C INTERFACE

#include <sunperf.h>

void ctrmm(char side, char uplo, char transa, char diag, int m, int n, complex alpha, complex *a, int lda, complex *b, int ldb);

void ctrmm_64(char side, char uplo, char transa, char diag, long m, long n, complex alpha, complex *a, long lda, complex *b, long ldb);

PURPOSE

ctrmm performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) multiplies B from the left or right as follows:

  SIDE = 'L' or 'l' B : = alpha*op(A)*B.

  SIDE = 'R' or 'r' B : = alpha*B*op(A).

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = conjg(A').

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity.

  Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1,M), when SIDE = 'R' or 'r' then LDA > = max(1,N). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading M by N part of the array B must contain the matrix B, and on exit is overwritten by the transformed matrix.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling subprogram. LDB must be at least max(1,M). Unchanged on exit.

NAME

ctrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE CTRMV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER N, LDA, INCY

SUBROUTINE CTRMV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TRMV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCY

SUBROUTINE TRMV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ctrmv(char uplo, char transa, char diag, int n, complex *a, int lda, complex *y, int incy);

void ctrmv_64(char uplo, char transa, char diag, long n, complex *a, long lda, complex *y, long incy);

PURPOSE

ctrmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE CTRRFS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)

 SUBROUTINE CTRRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*       X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LDX, INFO
 REAL FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TRRFS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, X
 INTEGER :: N, NRHS, LDA, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TRRFS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX, DIMENSION(:,:) :: A, B, X
 INTEGER(8) :: N, NRHS, LDA, LDB, LDX, INFO
 REAL, DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ctrrfs(char uplo, char transa, char diag, int n, int nrhs, complex *a, int lda, complex *b, int ldb, complex *x, int ldx, float *ferr, float *berr, int *info);

void ctrrfs_64(char uplo, char transa, char diag, long n, long nrhs, complex *a, long lda, complex *b, long ldb, complex *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

ctrrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix.

The solution matrix X must be computed by CTRTRS or some other means before entering this routine. CTRRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is

not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ctrsen - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace

---

SYNOPSIS

```
 SUBROUTINE CTRSEN( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, W, M, S,
*      SEP, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 COMPLEX T(LDT,*), Q(LDQ,*), W(*), WORK(*)
 INTEGER N, LDT, LDQ, M, LWORK, INFO
 LOGICAL SELECT(*)
 REAL S, SEP

 SUBROUTINE CTRSEN_64( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, W, M,
*      S, SEP, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 COMPLEX T(LDT,*), Q(LDQ,*), W(*), WORK(*)
 INTEGER*8 N, LDT, LDQ, M, LWORK, INFO
 LOGICAL*8 SELECT(*)
 REAL S, SEP
```

F95 INTERFACE

```
 SUBROUTINE TRSEN( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], W, M,
*       S, SEP, WORK, [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 COMPLEX, DIMENSION(:) :: W, WORK
 COMPLEX, DIMENSION(:,:) :: T, Q
 INTEGER :: N, LDT, LDQ, M, LWORK, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL :: S, SEP

 SUBROUTINE TRSEN_64( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], W,
*       M, S, SEP, WORK, [LWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOB, COMPQ
COMPLEX, DIMENSION(:) :: W, WORK
COMPLEX, DIMENSION(:,:) :: T, Q
INTEGER(8) :: N, LDT, LDQ, M, LWORK, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL :: S, SEP
```

C INTERFACE

#include <sunperf.h>

void ctrsen(char job, char compq, logical *select, int n, complex *t, int ldt, complex *q, int ldq, complex *w, int *m, float *s, float *sep, complex *work, int lwork, int *info);

void ctrsen_64(char job, char compq, logical *select, long n, complex *t, long ldt, complex *q, long ldq, complex *w, long *m, float *s, float *sep, complex *work, long lwork, long *info);

PURPOSE

ctrsen reorders the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (S) or the invariant subspace (SEP):

  ```
  = 'N': none;

  = 'E': for eigenvalues only (S);

  = 'V': for invariant subspace only (SEP);

  = 'B': for both eigenvalues and invariant subspace (S and
  SEP).
  ```
- **COMPQ (input)**

  ```
  = 'V': update the matrix Q of Schur vectors;

  = 'N': do not update Q.
  ```
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**

The order of the matrix T. N > = 0.

- **T (input/output)**
  On entry, the upper triangular matrix T. On exit, T is overwritten by the reordered matrix T, with the selected eigenvalues as the leading diagonal elements.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the unitary transformation matrix which reorders T; the leading M columns of Q form an orthonormal basis for the specified invariant subspace. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1; and if COMPQ = 'V', LDQ > = N.
- **W (output)**
  The reordered eigenvalues of T, in the same order as they appear on the diagonal of T.
- **M (output)**
  The dimension of the specified invariant subspace. 0 < = M < = N.
- **S (output)**
  If JOB = 'E' or 'B', S is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. S cannot underestimate the true reciprocal condition number by more than a factor of sqrt(N). If M = 0 or N, S = 1. If JOB = 'N' or 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', SEP is the estimated reciprocal condition number of the specified invariant subspace. If M = 0 or N, SEP = norm(T). If JOB = 'N' or 'E', SEP is not referenced.
- **WORK (output)**
  If JOB = 'N', WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOB = 'N', LWORK > = 1; if JOB = 'E', LWORK = M*(N-M); if JOB = 'V' or 'B', LWORK > = 2*M*(N-M).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

CTRSEN first collects the selected eigenvalues by computing a unitary transformation Z to move them to the top left corner of T. In other words, the selected eigenvalues are the eigenvalues of T11 in:

```
Z'*T*Z  = ( T11 T12 ) n1

          (  0  T22 ) n2

             n1  n2
```

where N = n1+n2 and Z' means the conjugate transpose of Z. The first n1 columns of Z span the specified invariant subspace

of T.

If T has been obtained from the Schur factorization of a matrix A = Q*T*Q', then the reordered Schur factorization of A is given by A = (Q*Z)*(Z'*T*Z)*(Q*Z)', and the first n1 columns of Q*Z span the corresponding invariant subspace of A.

The reciprocal condition number of the average of the eigenvalues of T11 may be returned in S. S lies between 0 (very badly conditioned) and 1 (very well conditioned). It is computed as follows. First we compute R so that

```
P  = ( I  R ) n1

     ( 0  0 ) n2

       n1 n2
```

is the projector on the invariant subspace associated with T11. R is the solution of the Sylvester equation:

```
T11*R - R*T22  = T12.
```

Let F-norm(M) denote the Frobenius-norm of M and 2-norm(M) denote the two-norm of M. Then S is computed as the lower bound

```
(1 + F-norm(R)**2)**(-1/2)
```

on the reciprocal of 2-norm(P), the true reciprocal condition number. S cannot underestimate 1 / 2-norm(P) by more than a factor of sqrt(N).

An approximate error bound for the computed average of the eigenvalues of T11 is

```
EPS * norm(T) / S
```

where EPS is the machine precision.

The reciprocal condition number of the right invariant subspace spanned by the first n1 columns of Z (or of Q*Z) is returned in SEP. SEP is defined as the separation of T11 and T22:

```
sep( T11, T22 )  = sigma-min( C )
```

where sigma-min(C) is the smallest singular value of the

n1*n2-by-n1*n2 matrix

```
C   = kprod( I(n2), T11 ) - kprod( transpose(T22), I(n1) )
```

`I(m)` is an m by m identity matrix, and kprod denotes the Kronecker product. We estimate sigma-min(C) by the reciprocal of an estimate of the 1-norm of inverse(C). The true reciprocal 1-norm of `inverse(C)` cannot differ from sigma-min(C) by more than a factor of sqrt(n1*n2).

When SEP is small, small changes in T can cause large changes in the invariant subspace. An approximate bound on the maximum angular error in the computed right invariant subspace is

```
EPS * norm(T) / SEP
```

NAME

ctrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

SYNOPSIS

```
 SUBROUTINE CTRSM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER M, N, LDA, LDB

 SUBROUTINE CTRSM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 COMPLEX ALPHA
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 M, N, LDA, LDB
```

F95 INTERFACE

```
 SUBROUTINE TRSM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: M, N, LDA, LDB

 SUBROUTINE TRSM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX :: ALPHA
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, LDA, LDB
```

C INTERFACE

#include <sunperf.h>

void ctrsm(char side, char uplo, char transa, char diag, int m, int n, complex alpha, complex *a, int lda, complex *b, int ldb);

void ctrsm_64(char side, char uplo, char transa, char diag, long m, long n, complex alpha, complex *a, long lda, complex *b, long ldb);

PURPOSE

ctrsm solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
```

The matrix X is overwritten on B.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) appears on the left or right of X as follows:

  SIDE = 'L' or 'l' op(A)*X = alpha*B.

  SIDE = 'R' or 'r' X*op(A) = alpha*B.

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = conjg(A').

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity.

  Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1,M), when SIDE = 'R' or 'r' then LDA > = max(1,N). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading M by N part of the array B must contain the right-hand side matrix B, and on exit is overwritten by the solution matrix X.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling subprogram. LDB > = max(1,M). Unchanged on exit.

NAME

ctrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary)

SYNOPSIS

```
 SUBROUTINE CTRSNA( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(LDWORK,*)
 INTEGER N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL SELECT(*)
 REAL S(*), SEP(*), WORK1(*)

 SUBROUTINE CTRSNA_64( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(LDWORK,*)
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL*8 SELECT(*)
 REAL S(*), SEP(*), WORK1(*)
```

F95 INTERFACE

```
 SUBROUTINE TRSNA( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 COMPLEX, DIMENSION(:,:) :: T, VL, VR, WORK
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, SEP, WORK1

 SUBROUTINE TRSNA_64( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 COMPLEX, DIMENSION(:,:) :: T, VL, VR, WORK
```

```
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL, DIMENSION(:) :: S, SEP, WORK1
```

C INTERFACE

#include <sunperf.h>

void ctrsna(char job, char howmny, logical *select, int n, complex *t, int ldt, complex *vl, int ldvl, complex *vr, int ldvr, float *s, float *sep, int mm, int *m, int ldwork, int *info);

void ctrsna_64(char job, char howmny, logical *select, long n, complex *t, long ldt, complex *vl, long ldvl, complex *vr, long ldvr, float *s, float *sep, long mm, long *m, long ldwork, long *info);

PURPOSE

ctrsna estimates reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary).

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (SEP):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (SEP);

  = 'B': for both eigenvalues and eigenvectors (S and SEP).
  ```
- **HOWMNY (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNY = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the j-th eigenpair, `SELECT(j)` must be set to .TRUE.. If HOWMNY = 'A', SELECT is not referenced.
- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input)**
  The upper triangular matrix T.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of T (or of any Q*T*Q**H with Q unitary), corresponding to

the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by CHSEIN or CTREVC. If JOB = 'V', VL is not referenced.

- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and if JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of T (or of any Q*T*Q**H with Q unitary), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by CHSEIN or CTREVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; and if JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. Thus S(j), SEP(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If JOB = 'E', SEP is not referenced.
- **MM (input)**
  The number of elements in the arrays S (if JOB = 'E' or 'B') and/or SEP (if JOB = 'V' or 'B'). MM > = M.
- **M (output)**
  The number of elements of the arrays S and/or SEP actually used to store the estimated condition numbers. If HOWMNY = 'A', M is set to N.
- **WORK (workspace)**
  `dimension(LDWORK,N+1)` If JOB = 'E', WORK is not referenced.
- **LDWORK (input)**
  The leading dimension of the array WORK. LDWORK > = 1; and if JOB = 'V' or 'B', LDWORK > = N.
- **WORK1 (workspace)**
  `dimension(N)` If JOB = 'E', WORK1 is not referenced.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of an eigenvalue lambda is defined as

```
S(lambda)  = |v'*u| / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of T corresponding to lambda; v' denotes the conjugate transpose of v, and `norm(u)` denotes the Euclidean norm. These reciprocal condition numbers always lie between zero (very badly conditioned) and one (very well conditioned). If n = 1, `S(lambda)` is defined to be 1.

An approximate error bound for a computed eigenvalue `W(i)` is given by

```
EPS * norm(T) / S(i)
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u corresponding to lambda is defined as follows. Suppose

```
T  = ( lambda   c   )

     (   0     T22 )
```

Then the reciprocal condition number is

```
SEP( lambda, T22 )  = sigma-min( T22 - lambda*I )
```

where sigma-min denotes the smallest singular value. We approximate the smallest singular value by the reciprocal of an estimate of the one-norm of the inverse of T22 - lambda*I. If n = 1, SEP(1) is defined to be abs(T(1,1)).

An approximate error bound for a computed right eigenvector VR(i) is given by

```
EPS * norm(T) / SEP(i)
```

NAME

ctrsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE CTRSV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER N, LDA, INCY

SUBROUTINE CTRSV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TRSV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCY

SUBROUTINE TRSV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX, DIMENSION(:) :: Y
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ctrsv(char uplo, char transa, char diag, int n, complex *a, int lda, complex *y, int incy);

void ctrsv_64(char uplo, char transa, char diag, long n, complex *a, long lda, complex *y, long incy);

PURPOSE

ctrsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**

On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ctrsyl - solve the complex Sylvester matrix equation

SYNOPSIS

```
 SUBROUTINE CTRSYL( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C, LDC,
*       SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER ISGN, M, N, LDA, LDB, LDC, INFO
 REAL SCALE

 SUBROUTINE CTRSYL_64( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C,
*       LDC, SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 ISGN, M, N, LDA, LDB, LDC, INFO
 REAL SCALE
```

F95 INTERFACE

```
 SUBROUTINE TRSYL( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL :: SCALE

 SUBROUTINE TRSYL_64( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 COMPLEX, DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL :: SCALE
```

C INTERFACE

#include <sunperf.h>

void ctrsyl(char trana, char tranb, int isgn, int m, int n, complex *a, int lda, complex *b, int ldb, complex *c, int ldc, float *scale, int *info);

void ctrsyl_64(char trana, char tranb, long isgn, long m, long n, complex *a, long lda, complex *b, long ldb, complex *c, long ldc, float *scale, long *info);

PURPOSE

ctrsyl solves the complex Sylvester matrix equation:

```
op(A)*X + X*op(B) = scale*C or

op(A)*X - X*op(B) = scale*C,
```

where `op(A)` = A or A**H, and A and B are both upper triangular. A is M-by-M and B is N-by-N; the right hand side C and the solution X are M-by-N; and scale is an output scale factor, set <= 1 to avoid overflow in X.

ARGUMENTS

- **TRANA (input)**
  Specifies the option op(A):

  ```
  = 'N': op(A)  = A    (No transpose)

  = 'C': op(A)  = A**H (Conjugate transpose)
  ```
- **TRANB (input)**
  Specifies the option op(B):

  ```
  = 'N': op(B)  = B    (No transpose)

  = 'C': op(B)  = B**H (Conjugate transpose)
  ```
- **ISGN (input)**
  Specifies the sign in the equation:

  ```
  = +1: solve op(A)*X + X*op(B)  = scale*C

  = -1: solve op(A)*X - X*op(B)  = scale*C
  ```
- **M (input)**
  The order of the matrix A, and the number of rows in the matrices X and C. M > = 0.
- **N (input)**
  The order of the matrix B, and the number of columns in the matrices X and C. N > = 0.
- **A (input)**

The upper triangular matrix A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input)**
  The upper triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **C (input/output)**
  On entry, the M-by-N right hand side matrix C. On exit, C is overwritten by the solution matrix X.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)`
- **SCALE (output)**
  The scale factor, scale, set <= 1 to avoid overflow in X.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1: A and B have common or very close eigenvalues; perturbed
values were used to solve the equation (but the matrices
A and B are unchanged).
```

NAME

ctrti2 - compute the inverse of a complex upper or lower triangular matrix

---

SYNOPSIS

```
SUBROUTINE CTRTI2( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE CTRTI2_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TRTI2( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE TRTI2_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ctrti2(char uplo, char diag, int n, complex *a, int lda, int *info);

void ctrti2_64(char uplo, char diag, long n, complex *a, long lda, long *info);

PURPOSE

ctrti2 computes the inverse of a complex upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **DIAG (input)**
  Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular

  ```
  = 'U':  Unit triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading n by n upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

  On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

ctrtri - compute the inverse of a complex upper or lower triangular matrix A

SYNOPSIS

```
SUBROUTINE CTRTRI( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE CTRTRI_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TRTRI( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE TRTRI_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ctrtri(char uplo, char diag, int n, complex *a, int lda, int *info);

void ctrtri_64(char uplo, char diag, long n, complex *a, long lda, long *info);

PURPOSE

ctrtri computes the inverse of a complex upper or lower triangular matrix A.

This is the Level 3 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1. On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value

> 0: if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

NAME

ctrtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE CTRTRS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NRHS, LDA, LDB, INFO

 SUBROUTINE CTRTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
 SUBROUTINE TRTRS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, INFO

 SUBROUTINE TRTRS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX, DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ctrtrs(char uplo, char transa, char diag, int n, int nrhs, complex *a, int lda, complex *b, int ldb, int *info);

void ctrtrs_64(char uplo, char transa, char diag, long n, long nrhs, complex *a, long lda, complex *b, long ldb, long *info);

PURPOSE

ctrtrs solves a triangular system of the form

where A is a triangular matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input/output)**

On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the solutions
X have not been computed.
```

NAME

ctzrqf - routine is deprecated and has been replaced by routine CTZRZF

SYNOPSIS

```
SUBROUTINE CTZRQF( M, N, A, LDA, TAU, INFO)
COMPLEX A(LDA,*), TAU(*)
INTEGER M, N, LDA, INFO

SUBROUTINE CTZRQF_64( M, N, A, LDA, TAU, INFO)
COMPLEX A(LDA,*), TAU(*)
INTEGER*8 M, N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TZRQF( [M], [N], A, [LDA], TAU, [INFO])
COMPLEX, DIMENSION(:) :: TAU
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO

SUBROUTINE TZRQF_64( [M], [N], A, [LDA], TAU, [INFO])
COMPLEX, DIMENSION(:) :: TAU
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ctzrqf(int m, int n, complex *a, int lda, complex *tau, int *info);

void ctzrqf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

ctzrqf routine is deprecated and has been replaced by routine CTZRZF.

CTZRQF reduces the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N unitary matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= M.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the unitary matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), whose conjugate transpose is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
Z( k )  = ( I     0   ),

          ( 0  T( k ) )
```

where

```
T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                              (   0    )
```

```
                                        ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

ctzrzf - reduce the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations

---

SYNOPSIS

```
SUBROUTINE CTZRZF( M, N, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LWORK, INFO

SUBROUTINE CTZRZF_64( M, N, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE TZRZF( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK, INFO

SUBROUTINE TZRZF_64( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void ctzrzf(int m, int n, complex *a, int lda, complex *tau, int *info);

void ctzrzf_64(long m, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

ctzrzf reduces the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N unitary matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the unitary matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

```
  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
   Z( k )  = ( I     0   ),

             ( 0  T( k ) )
```

where

```
   T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                                (   0    )
                                                ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

cung2l - generate an m by n complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE CUNG2L( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE CUNG2L_64( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNG2L( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNG2L_64( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cung2l(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cung2l_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cung2l L generates an m by n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by CGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cung2r - generate an m by n complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE CUNG2R( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE CUNG2R_64( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNG2R( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNG2R_64( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cung2r(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cung2r_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cung2r R generates an m by n complex matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by CGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cungbr - generate one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form

---

SYNOPSIS

```
SUBROUTINE CUNGBR( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE CUNGBR_64( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGBR( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGBR_64( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cungbr(char vect, int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungbr_64(char vect, long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungbr generates one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form: A = Q * B * P**H. Q and P**H are defined as products of elementary reflectors `H(i)` or `G(i)` respectively.

If VECT = 'Q', A is assumed to have been an M-by-K matrix, and Q is of order M:

if m >= k, Q = `H(1) H(2) . . . H(k)` and CUNGBR returns the first n columns of Q, where m >= n >= k;

if m < k, Q = `H(1) H(2) . . . H(m-1)` and CUNGBR returns Q as an M-by-M matrix.

If VECT = 'P', A is assumed to have been a K-by-N matrix, and P**H is of order N:

if k < n, P**H = `G(k) . . . G(2) G(1)` and CUNGBR returns the first m rows of P**H, where n >= m >= k;

if k >= n, P**H = `G(n-1) . . . G(2) G(1)` and CUNGBR returns P**H as an N-by-N matrix.

ARGUMENTS

- **VECT (input)**
  Specifies whether the matrix Q or the matrix P**H is required, as defined in the transformation applied by CGEBRD:

  ```
  = 'Q':  generate Q;

  = 'P':  generate P**H.
  ```
- **M (input)**
  The number of rows of the matrix Q or P**H to be returned. M > = 0.
- **N (input)**
  The number of columns of the matrix Q or P**H to be returned. N > = 0. If VECT = 'Q', M > = N > = min(M,K); if VECT = 'P', N > = M > = min(N,K).
- **K (input)**
  If VECT = 'Q', the number of columns in the original M-by-K matrix reduced by CGEBRD. If VECT = 'P', the number of rows in the original K-by-N matrix reduced by CGEBRD. K > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by CGEBRD. On exit, the M-by-N matrix Q or P**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = M.

- **TAU (input)**
  (min(M,K)) if VECT = 'Q' (min(N,K)) if VECT = 'P' `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or G(i), which determines Q or P**H, as returned by CGEBRD in its array argument TAUQ or TAUP.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,min(M,N)). For optimum performance LWORK >= min(M,N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunghr - generate a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD

SYNOPSIS

```
SUBROUTINE CUNGHR( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, ILO, IHI, LDA, LWORK, INFO

SUBROUTINE CUNGHR_64( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, ILO, IHI, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGHR( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, ILO, IHI, LDA, LWORK, INFO

SUBROUTINE UNGHR_64( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, ILO, IHI, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunghr(int n, int ilo, int ihi, complex *a, int lda, complex *tau, int *info);

void cunghr_64(long n, long ilo, long ihi, complex *a, long lda, complex *tau, long *info);

PURPOSE

cunghr generates a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **N (input)**
  The order of the matrix Q. N > = 0.
- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of CGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of IHI.
- **A (input)**
  On entry, the vectors which define the elementary reflectors, as returned by CGEHRD. On exit, the N-by-N unitary matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEHRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = IHI-ILO. For optimum performance LWORK > = (IHI-ILO)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cungl2 - generate an m-by-n complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE CUNGL2( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE CUNGL2_64( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGL2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNGL2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cungl2(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungl2_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungl2 generates an m-by-n complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

```
Q  =  H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  `= 0: successful exit`

  `< 0: if INFO  = -i, the i-th argument has an illegal value`

NAME

cunglq - generate an M-by-N complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE CUNGLQ( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE CUNGLQ_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGLQ( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGLQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunglq(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cunglq_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cunglq generates an M-by-N complex matrix Q with orthonormal rows, which is defined as the first M rows of a product of K elementary reflectors of order N

```
Q  =  H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit;

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cungql - generate an M-by-N complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE CUNGQL( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE CUNGQL_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGQL( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGQL_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cungql(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungql_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungql generates an M-by-N complex matrix Q with orthonormal columns, which is defined as the last N columns of a product of K elementary reflectors of order M

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by CGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. M >= N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N >= K >= 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N). For optimum performance LWORK >= N*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument has an illegal value
```

NAME

cungqr - generate an M-by-N complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE CUNGQR( M, N, K, A, LDA, TAU, WORKIN, LWORKIN, INFO)
COMPLEX A(LDA,*), TAU(*), WORKIN(*)
INTEGER M, N, K, LDA, LWORKIN, INFO

SUBROUTINE CUNGQR_64( M, N, K, A, LDA, TAU, WORKIN, LWORKIN, INFO)
COMPLEX A(LDA,*), TAU(*), WORKIN(*)
INTEGER*8 M, N, K, LDA, LWORKIN, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGQR( M, [N], [K], A, [LDA], TAU, [WORKIN], [LWORKIN],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORKIN
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORKIN, INFO

SUBROUTINE UNGQR_64( M, [N], [K], A, [LDA], TAU, [WORKIN], [LWORKIN],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORKIN
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORKIN, INFO
```

C INTERFACE

#include <sunperf.h>

void cungqr(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungqr_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungqr generates an M-by-N complex matrix Q with orthonormal columns, which is defined as the first N columns of a product of K elementary reflectors of order M

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by CGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The dimension of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cungr2 - generate an m by n complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE CUNGR2( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE CUNGR2_64( M, N, K, A, LDA, TAU, WORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGR2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNGR2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void cungr2(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungr2_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungr2 generates an m by n complex matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

```
Q  =  H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cungrq - generate an M-by-N complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE CUNGRQ( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE CUNGRQ_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGRQ( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGRQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cungrq(int m, int n, int k, complex *a, int lda, complex *tau, int *info);

void cungrq_64(long m, long n, long k, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungrq generates an M-by-N complex matrix Q with orthonormal rows, which is defined as the last M rows of a product of K elementary reflectors of order N

```
Q  =  H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. N > = M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M > = K > = 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,M). For optimum performance LWORK > = M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

cungtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD

SYNOPSIS

```
SUBROUTINE CUNGTR( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, LDA, LWORK, INFO

SUBROUTINE CUNGTR_64( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGTR( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER :: N, LDA, LWORK, INFO

SUBROUTINE UNGTR_64( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: TAU, WORK
COMPLEX, DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cungtr(char uplo, int n, complex *a, int lda, complex *tau, int *info);

void cungtr_64(char uplo, long n, complex *a, long lda, complex *tau, long *info);

PURPOSE

cungtr generates a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD:

if UPLO = 'U', Q = `H(n-1) . . . H(2)` H(1),

if UPLO = 'L', Q = `H(1) H(2) . . .` H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from CHETRD;
 = 'L': Lower triangle of A contains elementary reflectors
from CHETRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by CHETRD. On exit, the N-by-N unitary matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = N.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHETRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = N-1. For optimum performance LWORK > = (N-1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmbr - VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMBR( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMBR_64( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMBR( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMBR_64( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmbr(char vect, char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmbr_64(char vect, char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmbr VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N': Q * C C * Q TRANS = 'C': Q**H * C C * Q**H

If VECT = 'P', CUNMBR overwrites the general complex M-by-N matrix C with

```
                SIDE = 'L'     SIDE = 'R'
```

TRANS = 'N': P * C C * P

TRANS = 'C': P**H * C C * P**H

Here Q and P**H are the unitary matrices determined by CGEBRD when reducing a complex matrix A to bidiagonal form: A = Q * B * P**H. Q and P**H are defined as products of elementary reflectors `H(i)` and `G(i)` respectively.

Let nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Thus nq is the order of the unitary matrix Q or P**H that is applied.

If VECT = 'Q', A is assumed to have been an NQ-by-K matrix: if nq >= k, Q = `H(1) H(2)` . . . H(k);

if nq < k, Q = `H(1) H(2)` . . . H(nq-1).

If VECT = 'P', A is assumed to have been a K-by-NQ matrix: if k < nq, P = `G(1) G(2)` . . . G(k);

if k >= nq, P = `G(1) G(2)` . . . G(nq-1).

ARGUMENTS

- **VECT (input)**

  ```
  = 'Q': apply Q or Q**H;

  = 'P': apply P or P**H.
  ```

- **SIDE (input)**

  ```
  = 'L': apply Q, Q**H, P or P**H from the Left;

  = 'R': apply Q, Q**H, P or P**H from the Right.
  ```

- **TRANS (input)**

```
= 'N':  No transpose, apply Q or P;

= 'C':  Conjugate transpose, apply Q**H or P**H.
```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  If VECT = 'Q', the number of columns in the original matrix reduced by CGEBRD. If VECT = 'P', the number of rows in the original matrix reduced by CGEBRD. K >= 0.
- **A (input)**
  (LDA,min(nq,K)) if VECT = 'Q' (LDA,nq) if VECT = 'P' The vectors which define the elementary reflectors `H(i)` and G(i), whose products determine the matrices Q and P, as returned by CGEBRD.
- **LDA (input)**
  The leading dimension of the array A. If VECT = 'Q', LDA >= max(1,nq); if VECT = 'P', LDA >= max(1,min(nq,K)).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or `G(i)` which determines Q or P, as returned by CGEBRD in the array argument TAUQ or TAUP.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q or P*C or P**H*C or C*P or C*P**H.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmhr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMHR( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, ILO, IHI, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMHR_64( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, ILO, IHI, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMHR( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMHR_64( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmhr(char side, char trans, int m, int n, int ilo, int ihi, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmhr_64(char side, char trans, long m, long n, long ilo, long ihi, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmhr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of IHI-ILO elementary reflectors, as returned by CGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q**H (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of CGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). If SIDE = 'L', then 1 < = ILO < = IHI < = M, if M > 0, and ILO = 1 and IHI = 0, if M = 0; if SIDE = 'R', then 1 < = ILO < = IHI < = N, if N > 0, and ILO = 1 and IHI = 0, if N = 0.

- **IHI (input)**
  See the description of ILO.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CGEHRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**

(M-1) if SIDE = 'L' (N-1) if SIDE = 'R' TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGEHRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunml2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

---

SYNOPSIS

```
 SUBROUTINE CUNML2( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, INFO

 SUBROUTINE CUNML2_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNML2( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, INFO

 SUBROUTINE UNML2_64( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void cunml2(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunml2_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunml2 overwrites the general complex m-by-n matrix C with

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF. Q is of order m if SIDE = 'L' and of order n if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q' from the Left

  = 'R': apply Q or Q' from the Right
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q' (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.

- **C (input/output)**
  On entry, the m-by-n matrix C. On exit, C is overwritten by Q*C or Q'*C or C*Q' or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L', (M) if SIDE = 'R'
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmlq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMLQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMLQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMLQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMLQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmlq(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmlq_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmlq overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmql - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMQL( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMQL_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMQL( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMQL_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmql(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmql_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmql overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by CGEQLF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.
- **C (input/output)**

On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmqr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMQR( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMQR_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMQR( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMQR_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmqr(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmqr_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmqr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by CGEQRF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).

- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.

- **C (input/output)**

On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmr2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

---

SYNOPSIS

```
 SUBROUTINE CUNMR2( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, INFO

 SUBROUTINE CUNMR2_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMR2( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, INFO

 SUBROUTINE UNMR2_64( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmr2(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmr2_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmr2 overwrites the general complex m-by-n matrix C with

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF. Q is of order m if SIDE = 'L' and of order n if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q' from the Left

  = 'R': apply Q or Q' from the Right
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q' (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.

- **C (input/output)**
  On entry, the m-by-n matrix C. On exit, C is overwritten by Q*C or Q'*C or C*Q' or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L', (M) if SIDE = 'R'
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmrq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMRQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMRQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMRQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMRQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmrq(char side, char trans, int m, int n, int k, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmrq_64(char side, char trans, long m, long n, long k, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmrq overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M >= K >= 0; if SIDE = 'R', N >= K >= 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,K).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cunmrz - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, L, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, L, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE CUNMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*       WORK, LWORK, INFO)
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, L, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*       WORK, LWORK, INFO)
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, L, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmrz(char side, char trans, int m, int n, int k, int l, complex *a, int lda, complex *tau, complex *c, int ldc, complex *work, int lwork, int *info);

void cunmrz_64(char side, char trans, long m, long n, long k, long l, complex *a, long lda, complex *tau, complex *c, long ldc, complex *work, long lwork, long *info);

PURPOSE

cunmrz overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by CTZRZF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **L (input)**
  The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CTZRZF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,K).

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CTZRZF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

cunmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUNMTR( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, LDA, LDC, LWORK, INFO

 SUBROUTINE CUNMTR_64( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMTR( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER :: M, N, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMTR_64( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX, DIMENSION(:) :: TAU, WORK
 COMPLEX, DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void cunmtr(char side, char uplo, char trans, int m, int n, complex *a, int lda, complex *tau, complex *c, int ldc, int *info);

void cunmtr_64(char side, char uplo, char trans, long m, long n, complex *a, long lda, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cunmtr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by CHETRD:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **UPLO (input)**

  ```
  = 'U': Upper triangle of A contains elementary reflectors
from CHETRD;
  = 'L': Lower triangle of A contains elementary reflectors
from CHETRD.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CHETRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**
  (M-1) if SIDE = 'L' (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHETRD.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >=M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cupgtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by CHPTRD using packed storage

---

SYNOPSIS

```
SUBROUTINE CUPGTR( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), TAU(*), Q(LDQ,*), WORK(*)
INTEGER N, LDQ, INFO

SUBROUTINE CUPGTR_64( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
COMPLEX AP(*), TAU(*), Q(LDQ,*), WORK(*)
INTEGER*8 N, LDQ, INFO
```

F95 INTERFACE

```
SUBROUTINE UPGTR( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, TAU, WORK
COMPLEX, DIMENSION(:,:) :: Q
INTEGER :: N, LDQ, INFO

SUBROUTINE UPGTR_64( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX, DIMENSION(:) :: AP, TAU, WORK
COMPLEX, DIMENSION(:,:) :: Q
INTEGER(8) :: N, LDQ, INFO
```

C INTERFACE

#include <sunperf.h>

void cupgtr(char uplo, int n, complex *ap, complex *tau, complex *q, int ldq, int *info);

void cupgtr_64(char uplo, long n, complex *ap, complex *tau, complex *q, long ldq, long *info);

PURPOSE

cupgtr generates a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by CHPTRD using packed storage:

if UPLO = 'U', Q = `H(n-1)` . . . `H(2)` H(1),

if UPLO = 'L', Q = `H(1)` `H(2)` . . . H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangular packed storage used in previous
call to CHPTRD;
 = 'L': Lower triangular packed storage used in previous
call to CHPTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **AP (input)**
  The vectors which define the elementary reflectors, as returned by CHPTRD.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHPTRD.
- **Q (output)**
  The N-by-N unitary matrix Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **WORK (workspace)**
  `dimension(N-1)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

cupmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE CUPMTR( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 COMPLEX AP(*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, LDC, INFO

 SUBROUTINE CUPMTR_64( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 COMPLEX AP(*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UPMTR( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX, DIMENSION(:) :: AP, TAU, WORK
 COMPLEX, DIMENSION(:,:) :: C
 INTEGER :: M, N, LDC, INFO

 SUBROUTINE UPMTR_64( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX, DIMENSION(:) :: AP, TAU, WORK
 COMPLEX, DIMENSION(:,:) :: C
 INTEGER(8) :: M, N, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void cupmtr(char side, char uplo, char trans, int m, int n, complex *ap, complex *tau, complex *c, int ldc, int *info);

void cupmtr_64(char side, char uplo, char trans, long m, long n, complex *ap, complex *tau, complex *c, long ldc, long *info);

PURPOSE

cupmtr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by CHPTRD using packed storage:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
 = 'L': apply Q or Q**H from the Left;

 = 'R': apply Q or Q**H from the Right.
```

- **UPLO (input)**

```
 = 'U': Upper triangular packed storage used in previous
call to CHPTRD;
 = 'L': Lower triangular packed storage used in previous
call to CHPTRD.
```

- **TRANS (input)**

```
 = 'N':  No transpose, apply Q;

 = 'C':  Conjugate transpose, apply Q**H.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **AP (input)**
  (M*(M+1)/2) if SIDE = 'L' (N*(N+1)/2) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CHPTRD. AP is modified by the routine but restored on exit.
- **TAU (input)**
  or (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by

CHPTRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

vbrmm, svbrmm, dvbrmm, cvbrmm, zvbrmm - variable block sparse row format matrix-matrix multiply

---

SYNOPSIS

```
 SUBROUTINE SVBRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(KB+1),
*           BPNTRB(MB), BPNTRE(MB)
 REAL*4     ALPHA, BETA
 REAL*4     VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DVBRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(KB+1),
*           BPNTRB(MB), BPNTRE(MB)
 REAL*8     ALPHA, BETA
 REAL*8     VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CVBRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(KB+1),
*           BPNTRB(MB), BPNTRE(MB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZVBRMM( TRANSA, MB, N, KB, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, KB, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(KB+1),
*           BPNTRB(MB), BPNTRE(MB)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
         C <- alpha op(A) B + beta C
```

```
where ALPHA and BETA are scalar, C and B are  matrices,
A is a matrix represented in variable block sparse row  format
and op( A )  is one  of

op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
                                  ( ' indicates matrix transpose)
```

ARGUMENTS

```
TRANSA        Indicates how to operate with the sparse matrix
                0 : operate with matrix
                1 : operate with transpose matrix
                2 : operate with the conjugate transpose of matrix.
                    2 is equivalent to 1 if the matrix is real.

MB            Number of block rows in matrix A

N             Number of columns in matrix C

KB            Number of block columns in matrix A

ALPHA         Scalar parameter

DESCRA()      Descriptor argument.  Five element integer array
              DESCRA(1) matrix structure
                0 : general
                1 : symmetric (A=A')
                2 : Hermitian (A= CONJG(A'))
                3 : Triangular
                4 : Skew(Anti)-Symmetric (A=-A')
                5 : Diagonal
                6 : Skew-Hermitian (A= -CONJG(A'))
              DESCRA(2) upper/lower triangular indicator
                1 : lower
                2 : upper
              DESCRA(3) main diagonal type
                0 : non-unit
                1 : unit
              DESCRA(4) Array base  (NOT IMPLEMENTED)
                0 : C/C++ compatible
                1 : Fortran compatible
              DESCRA(5) repeated indices? (NOT IMPLEMENTED)
                0 : unknown
```

1 : no repeated indices

VAL()	scalar array of length NNZ consisting of the block entries of A where each block entry is a dense rectangular matrix stored column by column. NNZ is the total number of point entries in all nonzero block entries of a matrix A.
INDX()	integer array of length BNNZ+1 where BNNZ is the number of block entries of a matrix A such that the I-th element of INDX[] points to the location in VAL of the (1,1) element of the I-th block entry.
BINDX()	integer array of length BNNZ consisting of the block column indices of the block entries of A where BNNZ is the number block entries of a matrix A.
RPNTR()	integer array of length MB+1 such that RPNTR(I)-RPNTR(1)+1 is the row index of the first point row in the I-th block row. RPNTR(MB+1) is set to M+RPNTR(1) where M is the number of rows in matrix A. Thus, the number of point rows in the I-th block row is RPNTR(I+1)-RPNTR(I).
CPNTR()	integer array of length KB+1 such that CPNTR(J)-CPNTR(1)+1 is the column index of the first point column in the J-th block column. CPNTR(KB+1) is set to K+CPNTR(1) where K is the number of columns in matrix A. Thus, the number of point columns in the J-th block column is CPNTR(J+1)-CPNTR(J).
BPNTRB()	integer array of length MB such that BPNTRB(I)-BPNTRB(1)+1 points to location in BINDX of the first block entry of the I-th block row of A.
BPNTRE()	integer array of length MB such that BPNTRE(I)-BPNTRB(1) points to location in BINDX of the last block entry of the I-th block row of A.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. WORK is not referenced in the current version.

```
LWORK          length of WORK array. LWORK is not referenced
               in the current version.
```

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

It is known that there exits another representation of the variable block sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems", WPS, 1996). Its data structure consists of six array instead of the seven used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block row in the array BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of variable block sparse row format the following calling sequence should be used SUBROUTINE SVBRMM(TRANSA, MB, N, KB, ALPHA, DESCRA, * VAL, INDX, BINDX, RPNTR, CPNTR, IA, IA(2), * B, LDB, BETA, C, LDC, WORK, LWORK)

NAME

vbrsm, svbrsm, dvbrsm, cvbrsm, zvbrsm - variable block sparse row format triangular solve

---

SYNOPSIS

```
 SUBROUTINE SVBRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(MB+1),
*           BPNTRB(MB), BPNTRE(MB)
 REAL*4     ALPHA, BETA
 REAL*4     DV(*), VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE DVBRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(MB+1),
*           BPNTRB(MB), BPNTRE(MB)
 REAL*8     ALPHA, BETA
 REAL*8     DV(*), VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE CVBRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK )
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(MB+1),
*           BPNTRB(MB), BPNTRE(MB)
 COMPLEX*8  ALPHA, BETA
 COMPLEX*8  DV(*), VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)

 SUBROUTINE ZVBRSM( TRANSA, MB, N, UNITD, DV, ALPHA, DESCRA,
*           VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE,
*           B, LDB, BETA, C, LDC, WORK, LWORK)
 INTEGER*4  TRANSA, MB, N, UNITD, DESCRA(5), LDB, LDC, LWORK
 INTEGER*4  INDX(*), BINDX(*), RPNTR(MB+1), CPNTR(MB+1),
*           BPNTRB(MB), BPNTRE(MB)
 COMPLEX*16 ALPHA, BETA
 COMPLEX*16 DV(*), VAL(*), B(LDB,*), C(LDC,*), WORK(LWORK)
```

DESCRIPTION

```
  C <- ALPHA  op(A) B + BETA C     C <- ALPHA D op(A) B + BETA C
  C <- ALPHA op(A) D B + BETA C
```

where ALPHA and BETA are scalar, C and B are m by n dense matrices, D is a block diagonal matrix, A is a unit, or non-unit, upper or lower triangular matrix represented in variable block sparse row format and op(A) is one of

op(A) = inv(A) or op(A) = inv(A') or op(A) =inv(conjg(A'))
(inv denotes matrix inverse, ' indicates matrix transpose)

All blocks of A on the main diagonal MUST be triangular matrices.

=head1 ARGUMENTS

TRANSA	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix 2 : operate with the conjugate transpose of matrix. 2 is equivalent to 1 if matrix is real.
MB	Number of block rows in matrix A
N	Number of columns in matrix C
UNITD	Type of scaling: 1 : Identity matrix (argument DV[] is ignored) 2 : Scale on left (row block scaling) 3 : Scale on right (column block scaling)
DV()	Array containing the block entries of the block diagonal matrix D. The size of the J-th block is RPNTR(J+1)-RPNTR(J) and each block contains matrix entries stored column-major. The total length of array DV is given by the formula: sum over J from 1 to MB: ((RPNTR(J+1)-RPNTR(J))*(RPNTR(J+1)-RPNTR(J)))
ALPHA	Scalar parameter
DESCRA()	Descriptor argument. Five element integer array DESCRA(1) matrix structure 0 : general 1 : symmetric (A=A') 2 : Hermitian (A= CONJG(A')) 3 : Triangular 4 : Skew(Anti)-Symmetric (A=-A')

5 : Diagonal
6 : Skew-Hermitian (A= -CONJG(A'))
Note: For the routine, DESCRA(1)=3 is only supported.

DESCRA(2) upper/lower triangular indicator
1 : lower
2 : upper
DESCRA(3) main diagonal type
0 : non-identity blocks on the main diagonal
1 : identity diagonal block
DESCRA(4) Array base (NOT IMPLEMENTED)
0 : C/C++ compatible
1 : Fortran compatible
DESCRA(5) repeated indices? (NOT IMPLEMENTED)
0 : unknown
1 : no repeated indices

VAL() scalar array of length NNZ consisting of the block entries of A where each block entry is a dense rectangular matrix stored column by column.
NNZ is the total number of point entries in all nonzero block entries of a matrix A.

INDX() integer array of length BNNZ+1 where BNNZ is the number block entries of a matrix A such that the I-th element of INDX[] points to the location in VAL of the (1,1) element of the I-th block entry.

BINDX() integer array of length BNNZ consisting of the block column indices of the block entries of A where BNNZ is the number block entries of a matrix A. Block column indices MUST be sorted in increasing order for each block row.

RPNTR() integer array of length MB+1 such that RPNTR(I)-RPNTR(1)+1 is the row index of the first point row in the I-th block row.
RPNTR(MB+1) is set to M+RPNTR(1) where M is the number of rows in square triangular matrix A.
Thus, the number of point rows in the I-th block row is RPNTR(I+1)-RPNTR(I).

NOTE: For the current version CPNTR must equal RPNTR and a single array can be passed for both arguments

CPNTR() integer array of length MB+1 such that CPNTR(J)-CPNTR(1)+1 is the column index of the first point column in the J-th block column. CPNTR(MB+1) is set to M+CPNTR(1).
Thus, the number of point columns in the J-th block column is CPNTR(J+1)-CPNTR(J).

NOTE: For the current version CPNTR must equal RPNTR and a single array can be passed for both arguments

BPNTRB() integer array of length MB such that BPNTRB(I)-BPNTRB(1)+1 points to location in BINDX of the first block entry of the I-th block row of A.

BPNTRE()	integer array of length MB such that BPNTRE(I)-BPNTRB(1) points to location in BINDX of the last block entry of the I-th block row of A.
B()	rectangular array with first dimension LDB.
LDB	leading dimension of B
BETA	Scalar parameter
C()	rectangular array with first dimension LDC.
LDC	leading dimension of C
WORK()	scratch array of length LWORK. On exit, if LWORK= -1, WORK(1) returns the optimum size of LWORK.
LWORK	length of WORK array. LWORK should be at least M = RPNTR(MB+1)-RPNTR(1). For good performance, LWORK should generally be larger. For optimum performance on multiple processors, LWORK >=M*N_CPUS where N_CPUS is the maximum number of processors available to the program. If LWORK=0, the routine is to allocate workspace needed. If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimum size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

SEE ALSO

NIST FORTRAN Sparse Blas User's Guide available at:

http://math.nist.gov/mcsd/Staff/KRemington/fspblas/

NOTES/BUGS

1. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

2. It is known that there exits another representation of the variable block sparse row format (see for example Y.Saad, ``Iterative Methods for Sparse Linear Systems'', WPS, 1996). Its data structure consists of six array instead of the seven used in the current implementation. The main difference is that only one array, IA, containing the pointers to the beginning of each block row in the array BINDX is used instead of two arrays BPNTRB and BPNTRE. To use the routine with this kind of variable block sparse row format the following calling sequence should be used SUBROUTINE SVBRMM(TRANSA, MB, N, KB, ALPHA, DESCRA, * VAL, INDX, BINDX, RPNTR, CPNTR, IA, IA(2), * B, LDB, BETA, C, LDC, WORK, LWORK)

NAME

cvmul - compute the scaled product of complex vectors

SYNOPSIS

```
SUBROUTINE CVMUL( N, ALPHA, X, INCX, Y, INCY, BETA, Z, INCZ)
COMPLEX ALPHA, BETA
COMPLEX X(*), Y(*), Z(*)
INTEGER N, INCX, INCY, INCZ

SUBROUTINE CVMUL_64( N, ALPHA, X, INCX, Y, INCY, BETA, Z, INCZ)
COMPLEX ALPHA, BETA
COMPLEX X(*), Y(*), Z(*)
INTEGER*8 N, INCX, INCY, INCZ
```

F95 INTERFACE

```
SUBROUTINE VMUL( [N], ALPHA, X, [INCX], Y, [INCY], BETA, Z, [INCZ])
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: X, Y, Z
INTEGER :: N, INCX, INCY, INCZ

SUBROUTINE VMUL_64( [N], ALPHA, X, [INCX], Y, [INCY], BETA, Z, [INCZ])
COMPLEX :: ALPHA, BETA
COMPLEX, DIMENSION(:) :: X, Y, Z
INTEGER(8) :: N, INCX, INCY, INCZ
```

C INTERFACE

#include <sunperf.h>

void cvmul(int n, complex alpha, complex *x, int incx, complex *y, int incy, complex beta, complex *z, int incz);

void cvmul_64(long n, complex alpha, complex *x, long incx, complex *y, long incy, complex beta, complex *z, long incz);

PURPOSE

cvmul computes the scaled product of complex vectors:

```
z(i) = ALPHA * x(i) * y(i) + BETA * z(i)
```

for 1 <= i <= N.

ARGUMENTS

- **N (input)**
  Length of the vectors. N > = 0. Returns immediately if N = 0.
- **ALPHA (input)**
  Scale factor on the multiplicand vectors.
- **X (input)**

  ```
  dimension(*)
  ```

  Multiplicand vector.
- **INCX (input)**
  Stride between elements of the multiplicand vector X. INCX > 0.
- **Y (input)**

  ```
  dimension(*)
  ```

  Multiplicand vector.
- **INCY (input)**
  Stride between elements of the multiplicand vector Y. INCY > 0.
- **BETA (input)**
  Scale factor on the product vector.
- **Z (input/output)**

  ```
  dimension(*)
  ```

  Product vector. On exit, `z(i)` = ALPHA * `x(i)` * `y(i)` + BETA * z(i).
- **INCZ (input)**
  Stride between elements of Z. INCZ > 0.

NAME

dasum - Return the sum of the absolute values of a vector x.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DASUM( N, X, INCX)
INTEGER N, INCX
DOUBLE PRECISION X(*)

DOUBLE PRECISION FUNCTION DASUM_64( N, X, INCX)
INTEGER*8 N, INCX
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION ASUM( [N], X, [INCX])
INTEGER :: N, INCX
REAL(8), DIMENSION(:) :: X

REAL(8) FUNCTION ASUM_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

double dasum(int n, double *x, int incx);

double dasum_64(long n, double *x, long incx);

PURPOSE

dasum Return the sum of the absolute values of x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.

NAME

daxpy - compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE DAXPY( N, ALPHA, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*)

SUBROUTINE DAXPY_64( N, ALPHA, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE AXPY( [N], ALPHA, X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y

SUBROUTINE AXPY_64( [N], ALPHA, X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void daxpy(int n, double alpha, double *x, int incx, double *y, int incy);

void daxpy_64(long n, double alpha, double *x, long incx, double *y, long incy);

PURPOSE

daxpy compute y := alpha * x + y where alpha is a scalar and x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

daxpyi - Compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE DAXPYI(NZ, A, X, INDX, Y)

DOUBLE PRECISION A
DOUBLE PRECISION X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE DAXPYI_64(NZ, A, X, INDX, Y)

DOUBLE PRECISION A
DOUBLE PRECISION X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE AXPYI([NZ], [A], X, INDX, Y)

```
REAL(8) :: A
REAL(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE AXPYI_64([NZ], [A], X, INDX, Y)

REAL(8) :: A
REAL(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

DAXPYI Compute y := alpha * x + y where alpha is a scalar, x is a sparse vector, and y is a vector in full storage form

```
do i = 1, n
  y(indx(i)) = alpha * x(i) + y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

A (input)

On entry, ALPHA specifies the scaling value. Unchanged on exit.

X (input)

Vector containing the values of the compressed form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

NAME

dbdsdc - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B

SYNOPSIS

```
 SUBROUTINE DBDSDC( UPLO, COMPQ, N, D, E, U, LDU, VT, LDVT, Q, IQ,
*      WORK, IWORK, INFO)
 CHARACTER * 1 UPLO, COMPQ
 INTEGER N, LDU, LDVT, INFO
 INTEGER IQ(*), IWORK(*)
 DOUBLE PRECISION D(*), E(*), U(LDU,*), VT(LDVT,*), Q(*), WORK(*)

 SUBROUTINE DBDSDC_64( UPLO, COMPQ, N, D, E, U, LDU, VT, LDVT, Q, IQ,
*      WORK, IWORK, INFO)
 CHARACTER * 1 UPLO, COMPQ
 INTEGER*8 N, LDU, LDVT, INFO
 INTEGER*8 IQ(*), IWORK(*)
 DOUBLE PRECISION D(*), E(*), U(LDU,*), VT(LDVT,*), Q(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSDC( UPLO, COMPQ, [N], D, E, U, [LDU], VT, [LDVT], Q,
*       IQ, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO, COMPQ
 INTEGER :: N, LDU, LDVT, INFO
 INTEGER, DIMENSION(:) :: IQ, IWORK
 REAL(8), DIMENSION(:) :: D, E, Q, WORK
 REAL(8), DIMENSION(:,:) :: U, VT

 SUBROUTINE BDSDC_64( UPLO, COMPQ, [N], D, E, U, [LDU], VT, [LDVT],
*       Q, IQ, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO, COMPQ
 INTEGER(8) :: N, LDU, LDVT, INFO
 INTEGER(8), DIMENSION(:) :: IQ, IWORK
 REAL(8), DIMENSION(:) :: D, E, Q, WORK
 REAL(8), DIMENSION(:,:) :: U, VT
```

C INTERFACE

#include <sunperf.h>

void dbdsdc(char uplo, char compq, int n, double *d, double *e, double *u, int ldu, double *vt, int ldvt, double *q, int *iq, int *info);

void dbdsdc_64(char uplo, char compq, long n, double *d, double *e, double *u, long ldu, double *vt, long ldvt, double *q, long *iq, long *info);

PURPOSE

dbdsdc computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = U * S * VT, using a divide and conquer method, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and VT are orthogonal matrices of left and right singular vectors, respectively. SBDSDC can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLASD3 for details.

The code currently call SLASDQ if singular values only are desired. However, it can be slightly modified to compute singular values using the divide and conquer method.

ARGUMENTS

- **UPLO (input)**

```
  = 'U':  B is upper bidiagonal.

  = 'L':  B is lower bidiagonal.
```

- **COMPQ (input)**
  Specifies whether singular vectors are to be computed as follows:

```
  = 'N':  Compute singular values only;

  = 'P':  Compute singular values and compute singular
vectors in compact form;
  = 'I':  Compute singular values and singular vectors.
```

- **N (input)**
  The order of the matrix B. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of the bidiagonal matrix whose SVD is desired. On exit, E has been destroyed.

- **U (output)**
  If COMPQ = 'I', then: On exit, if INFO = 0, U contains the left singular vectors of the bidiagonal matrix. For other values of COMPQ, U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU > = 1. If singular vectors are desired, then LDU > = max(1, N).
- **VT (output)**
  If COMPQ = 'I', then: On exit, if INFO = 0, VT' contains the right singular vectors of the bidiagonal matrix. For other values of COMPQ, VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT > = 1. If singular vectors are desired, then LDVT > = max(1, N).
- **Q (output)**
  If COMPQ = 'P', then: On exit, if INFO = 0, Q and IQ contain the left and right singular vectors in a compact form, requiring O(N log N) space instead of 2*N**2. In particular, Q contains all the REAL data in LDQ > = N*(11 + 2*SMLSIZ + 8*INT(LOG_2(N/(SMLSIZ+1)))) words of memory, where SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25). For other values of COMPQ, Q is not referenced.
- **IQ (output)**
  If COMPQ = 'P', then: On exit, if INFO = 0, Q and IQ contain the left and right singular vectors in a compact form, requiring O(N log N) space instead of 2*N**2. In particular, IQ contains all INTEGER data in LDIQ > = N*(3 + 3*INT(LOG_2(N/(SMLSIZ+1)))) words of memory, where SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25). For other values of COMPQ, IQ is not referenced.
- **WORK (workspace)**
  If COMPQ = 'N' then LWORK > = (2 * N). If COMPQ = 'P' then LWORK > = (6 * N). If COMPQ = 'I' then LWORK > = (3 * N**2 + 4 * N).
- **IWORK (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an singular value.
The update process of divide and conquer failed.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Huan Ren, Computer Science Division, University of
California at Berkeley, USA
```

NAME

dbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

SYNOPSIS

```
 SUBROUTINE DBDSQR( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U, LDU,
*      C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 DOUBLE PRECISION D(*), E(*), VT(LDVT,*), U(LDU,*), C(LDC,*), WORK(*)

 SUBROUTINE DBDSQR_64( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U,
*      LDU, C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 DOUBLE PRECISION D(*), E(*), VT(LDVT,*), U(LDU,*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSQR( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT, [LDVT],
*       U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: VT, U, C

 SUBROUTINE BDSQR_64( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT,
*       [LDVT], U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: VT, U, C
```

C INTERFACE

#include <sunperf.h>

void dbdsqr(char uplo, int n, int ncvt, int nru, int ncc, double *d, double *e, double *vt, int ldvt, double *u, int ldu, double *c, int ldc, int *info);

void dbdsqr_64(char uplo, long n, long ncvt, long nru, long ncc, double *d, double *e, double *vt, long ldvt, double *u, long ldu, double *c, long ldc, long *info);

PURPOSE

dbdsqr computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = Q * S * P' (P' denotes the transpose of P), where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and Q and P are orthogonal matrices.

The routine computes S, and optionally computes U * Q, P' * VT, or Q' * C, for given real input matrices U, VT, and C.

See ``Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy," by J. Demmel and W. Kahan, LAPACK Working Note #3 (or SIAM J. Sci. Statist. Comput. vol. 11, no. 5, pp. 873-912, Sept 1990) and

``Accurate singular values and differential qd algorithms," by B. Parlett and V. Fernando, Technical Report CPAM-554, Mathematics Department, University of California at Berkeley, July 1992 for a detailed description of the algorithm.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  B is upper bidiagonal;

  = 'L':  B is lower bidiagonal.
  ```

- **N (input)**
  The order of the matrix B. N > = 0.
- **NCVT (input)**
  The number of columns of the matrix VT. NCVT > = 0.
- **NRU (input)**
  The number of rows of the matrix U. NRU > = 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B in decreasing order.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of the bidiagonal matrix whose SVD is desired. On normal exit (INFO = 0), E is destroyed. If the algorithm does not converge (INFO > 0), D and E will contain the diagonal and superdiagonal elements of a bidiagonal matrix orthogonally equivalent to the one given as input. E(N) is used for workspace.
- **VT (input/output)**

On entry, an N-by-NCVT matrix VT. On exit, VT is overwritten by P' * VT. VT is not referenced if NCVT = 0.

- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= `max(1,N)` if NCVT > 0; LDVT >= 1 if NCVT = 0.
- **U (input/output)**
  On entry, an NRU-by-N matrix U. On exit, U is overwritten by U * Q. U is not referenced if NRU = 0.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,NRU).
- **C (input/output)**
  On entry, an N-by-NCC matrix C. On exit, C is overwritten by Q' * C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,N)` if NCC > 0; LDC >=1 if NCC = 0.
- **WORK (workspace)**
  `dimension(4*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  If INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm did not converge; D and E contain the
elements of a bidiagonal matrix which is orthogonally
similar to the input matrix B;  if INFO  = i, i
elements of E have not converged to zero.
```

NAME

dcnvcor - compute the convolution or correlation of real vectors

---

SYNOPSIS

```
 SUBROUTINE DCNVCOR( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M, Y,
*      IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 INTEGER NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 DOUBLE PRECISION X(*), Y(*), Z(*), WORK(*)

 SUBROUTINE DCNVCOR_64( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M,
*      Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 INTEGER*8 NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 DOUBLE PRECISION X(*), Y(*), Z(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE, M,
*       Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 INTEGER :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL(8), DIMENSION(:) :: X, Y, Z, WORK

 SUBROUTINE CNVCOR_64( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE,
*       M, Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK,
*       [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 INTEGER(8) :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL(8), DIMENSION(:) :: X, Y, Z, WORK
```

C INTERFACE

#include <sunperf.h>

void dcnvcor(char cnvcor, char four, int nx, double *x, int ifx, int incx, int ny, int npre, int m, double *y, int ify, int inc1y, int inc2y, int nz, int k, double *z, int ifz, int inc1z, int inc2z, double *work, int lwork);

void dcnvcor_64(char cnvcor, char four, long nx, double *x, long ifx, long incx, long ny, long npre, long m, double *y, long ify, long inc1y, long inc2y, long nz, long k, double *z, long ifz, long inc1z, long inc2z, double *work, long lwork);

PURPOSE

dcnvcor computes the convolution or correlation of real vectors.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' if convolution is desired, 'R' or 'r' if correlation is desired.
- **FOUR (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' if the computation should be done directly from the definition. The Fourier transform method is generally faster, but it may introduce noticeable errors into certain results, notably when both the filter and data vectors consist entirely of integers or vectors where elements of either the filter vector or a given data vector differ significantly in magnitude from the 1-norm of the vector.
- **NX (input)**
  Length of the filter vector. NX > = 0. DCNVCOR will return immediately if NX = 0.
- **X (input)**
  Filter vector.
- **IFX (input)**
  Index of the first element of X. NX > = IFX > = 1.
- **INCX (input)**
  Stride between elements of the filter vector in X. INCX > 0.
- **NY (input)**
  Length of the input vectors. NY > = 0. DCNVCOR will return immediately if NY = 0.
- **NPRE (input)**
  The number of implicit zeros prepended to the Y vectors. NPRE > = 0.
- **M (input)**
  Number of input vectors. M > = 0. DCNVCOR will return immediately if M = 0.
- **Y (input)**
  Input vectors.
- **IFY (input)**
  Index of the first element of Y. NY > = IFY > = 1.
- **INC1Y (input)**
  Stride between elements of the input vectors in Y. INC1Y > 0.
- **INC2Y (input)**
  Stride between the input vectors in Y. INC2Y > 0.
- **NZ (input)**
  Length of the output vectors. NZ > = 0. DCNVCOR will return immediately if NZ = 0. See the Notes section below for information about how this argument interacts with NX and NY to control circular versus end-off shifting.
- **K (input)**
  Number of Z vectors. K > = 0. If K = 0 then DCNVCOR will return immediately. If K < M then only the first K input vectors will be processed. If K > M then M input vectors will be processed.
- **Z (output)**
  Result vectors.
- **IFZ (input)**
  Index of the first element of Z. NZ > = IFZ > = 1.
- **INC1Z (input)**
  Stride between elements of the output vectors in Z. INC1Z > 0.
- **INC2Z (input)**
  Stride between the output vectors in Z. INC2Z > 0.
- **WORK (input/output)**
  Scratch space. Before the first call to DCNVCOR with particular values of the integer arguments the first element of WORK must be set to zero. If WORK is written between calls to DCNVCOR or if DCNVCOR is called with different values of the integer arguments then the first element of WORK must again be set to zero before each call. If WORK has not been written and the same

values of the integer arguments are used then the first element of WORK to zero. This can avoid certain initializations that store their results into WORK, and avoiding the initialization can make DCNVCOR run faster.

- **LWORK (input)**
  Length of WORK. LWORK > = 4*MAX(NX,NY,NZ)+15. =head1 NOTES If any vector overlaps a writable vector, either because of argument aliasing or ill-chosen values of the various INC arguments, the results are undefined and may vary from one run to the next.

  The most common form of the computation, and the case that executes fastest, is applying a filter vector X to a series of vectors stored in the columns of Y with the result placed into the columns of Z. In that case, INCX = 1, INC1Y = 1, INC2Y > = NY, INC1Z = 1, INC2Z > = NZ. Another common form is applying a filter vector X to a series of vectors stored in the rows of Y and store the result in the row of Z, in which case INCX = 1, INC1Y > = NY, INC2Y = 1, INC1Z > = NZ, and INC2Z = 1.

  A common use of convolution is to compute the products of polynomials. The following code uses DCNVCOR to compute the product of 1 + 2x + 3x**2 and 4 + 5x + 6x**2:

NAME

dcnvcor2 - compute the convolution or correlation of real matrices

---

SYNOPSIS

```
 SUBROUTINE DCNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*       LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 DOUBLE COMPLEX WORKIN(*)
 INTEGER MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 DOUBLE PRECISION X(LDX,*), Y(LDY,*), Z(LDZ,*)

 SUBROUTINE DCNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*       LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 DOUBLE COMPLEX WORKIN(*)
 INTEGER*8 MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 DOUBLE PRECISION X(LDX,*), Y(LDY,*), Z(LDZ,*)
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX(8), DIMENSION(:) :: WORKIN
 INTEGER :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL(8), DIMENSION(:,:) :: X, Y, Z

 SUBROUTINE CNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX(8), DIMENSION(:) :: WORKIN
 INTEGER(8) :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL(8), DIMENSION(:,:) :: X, Y, Z
```

C INTERFACE

#include <sunperf.h>

void dcnvcor2(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, int mx, int nx, double *x, int ldx, int my, int ny, int mpre, int npre, double *y, int ldy, int mz, int nz, double *z, int ldz, doublecomplex *workin, int lwork);

void dcnvcor2_64(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, long mx, long nx, double *x, long ldx, long my, long ny, long mpre, long npre, double *y, long ldy, long mz, long nz, double *z, long ldz, doublecomplex *workin, long lwork);

PURPOSE

dcnvcor2 computes the convolution or correlation of real matrices.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' to compute convolution, 'R' or 'r' to compute correlation.
- **METHOD (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' to compute directly from the definition.
- **TRANSX (input)**
  \'N' or 'n' if X is the filter matrix, 'T' or 't' if `transpose(X)` is the filter matrix.
- **SCRATCHX (input)**
  \'N' or 'n' if X must be preserved, 'S' or 's' if X can be used as scratch space. The contents of X are undefined after returning from a call in which X is allowed to be used for scratch.
- **TRANSY (input)**
  \'N' or 'n' if Y is the input matrix, 'T' or 't' if `transpose(Y)` is the input matrix.
- **SCRATCHY (input)**
  \'N' or 'n' if Y must be preserved, 'S' or 's' if Y can be used as scratch space. The contents of Y are undefined after returning from a call in which Y is allowed to be used for scratch.
- **MX (input)**
  Number of rows in the filter matrix. MX > = 0.
- **NX (input)**
  Number of columns in the filter matrix. NX > = 0.
- **X (input)**

  ```
  dimension(LDX,NX)
  ```

  On entry, the filter matrix. Unchanged on exit if SCRATCHX is 'N' or 'n', undefined on exit if SCRATCHX is 'S' or 's'.
- **LDX (input)**
  Leading dimension of the array that contains the filter matrix.
- **MY (input)**
  Number of rows in the input matrix. MY > = 0.

- **NY (input)**
  Number of columns in the input matrix. NY > = 0.
- **MPRE (input)**
  Number of implicit zeros to prepend to each row of the input matrix. MPRE > = 0.
- **NPRE (input)**
  Number of implicit zeros to prepend to each column of the input matrix. NPRE > = 0.
- **Y (input)**

  ```
  dimension(LDY,*)
  ```

  Input matrix. Unchanged on exit if SCRATCHY is 'N' or 'n', undefined on exit if SCRATCHY is 'S' or 's'.
- **LDY (input)**
  Leading dimension of the array that contains the input matrix.
- **MZ (input)**
  Number of rows in the output matrix. MZ > = 0. DCNVCOR2 will return immediately if MZ = 0.
- **NZ (input)**
  Number of columns in the output matrix. NZ > = 0. DCNVCOR2 will return immediately if NZ = 0.
- **Z (output)**

  ```
  dimension(LDZ,*)
  ```

  Result matrix.
- **LDZ (input)**
  Leading dimension of the array that contains the result matrix. LDZ > = MAX(1,MZ).
- **WORKIN (input/output)**
  (input/scratch) `dimension(LWORK)`

  On entry for the first call to DCNVCOR2, `WORKIN(1)` must contain 0.0. After the first call, `WORKIN(1)` must be set to 0.0 iff WORKIN has been altered since the last call to this subroutine or if the sizes of the arrays have changed.
- **LWORK (input)**
  Length of the work vector. If the FFT is to be used then for best performance LWORK should be at least 30 words longer than the amount of memory needed to hold the trig tables. If the FFT is not used, the value of LWORK is unimportant.

NAME

dcopy - Copy x to y

SYNOPSIS

```
SUBROUTINE DCOPY( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)

SUBROUTINE DCOPY_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE COPY( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y

SUBROUTINE COPY_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void dcopy(int n, double *x, int incx, double *y, int incy);

void dcopy_64(long n, double *x, long incx, double *y, long incy);

PURPOSE

dcopy Copy x to y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

dcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE DCOSQB( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DCOSQB_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQB( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE COSQB_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dcosqb(int n, double *x, double *wsave);

void dcosqb_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave cosine synthesis of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) that has been initialized by DCOSQI.

NAME

dcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The COSQ operations are unnormalized inverses of themselves, so a call to COSQF followed by a call to COSQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE DCOSQF( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DCOSQF_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQF( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE COSQF_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dcosqf(int n, double *x, double *wsave);

void dcosqf_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave cosine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) that has been initialized by DCOSQI.

NAME

dcosqi - initialize the array WSAVE, which is used in both COSQF and COSQB.

SYNOPSIS

```
SUBROUTINE DCOSQI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DCOSQI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE COSQI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dcosqi(int n, double *wsave);

void dcosqi_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. DCOSQI needs to be called only once to initialize WSAVE before calling DCOSQF and/or DCOSQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dcost - compute the discrete Fourier cosine transform of an even sequence. The COST transforms are unnormalized inverses of themselves, so a call of COST followed by another call of COST will multiply the input sequence by 2 * (N-1).

SYNOPSIS

```
SUBROUTINE DCOST( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DCOST_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COST( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE COST_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dcost(int n, double *x, double *wsave);

void dcost_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N - 1 is a product of small primes. N > = 2.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the cosine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by DCOSTI.

NAME

dcosti - initialize the array WSAVE, which is used in COST.

SYNOPSIS

```
SUBROUTINE DCOSTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DCOSTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE COSTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dcosti(int n, double *wsave);

void dcosti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N - 1 is a product of small primes. N >
  = 2.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. DCOSTI is called once to initialize WSAVE before calling DCOST and need not be called again between calls to DCOST if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

ddisna - compute the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix

---

SYNOPSIS

```
SUBROUTINE DDISNA( JOB, M, N, D, SEP, INFO)
CHARACTER * 1 JOB
INTEGER M, N, INFO
DOUBLE PRECISION D(*), SEP(*)

SUBROUTINE DDISNA_64( JOB, M, N, D, SEP, INFO)
CHARACTER * 1 JOB
INTEGER*8 M, N, INFO
DOUBLE PRECISION D(*), SEP(*)
```

F95 INTERFACE

```
SUBROUTINE DISNA( JOB, [M], N, D, SEP, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER :: M, N, INFO
REAL(8), DIMENSION(:) :: D, SEP

SUBROUTINE DISNA_64( JOB, [M], N, D, SEP, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER(8) :: M, N, INFO
REAL(8), DIMENSION(:) :: D, SEP
```

C INTERFACE

#include <sunperf.h>

void ddisna(char job, int m, int n, double *d, double *sep, int *info);

void ddisna_64(char job, long m, long n, double *d, double *sep, long *info);

PURPOSE

ddisna computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix. The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the I-th computed vector is given by

```
SLAMCH( 'E' ) * ( ANORM / SEP( I ) )
```

where ANORM = 2-norm(A) = max(abs(D(j))). SEP(I) is not allowed to be smaller than SLAMCH('E')*ANORM in order to limit the size of the error bound.

SDISNA may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

ARGUMENTS

- **JOB (input)**
  Specifies for which problem the reciprocal condition numbers should be computed:

  ```
  = 'E':  the eigenvectors of a symmetric/Hermitian matrix;

  = 'L':  the left singular vectors of a general matrix;

  = 'R':  the right singular vectors of a general matrix.
  ```
- **M (input)**
  The number of rows of the matrix. M >= 0.
- **N (input)**
  If JOB = 'L' or 'R', the number of columns of the matrix, in which case N >= 0. Ignored if JOB = 'E'.
- **D (input)**
  dimension (min(M,N)) if JOB = 'L' or 'R' The eigenvalues (if JOB = 'E') or singular values (if JOB = 'L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.
- **SEP (output)**
  dimension (min(M,N)) if JOB = 'L' or 'R' The reciprocal condition numbers of the vectors.
- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

ddot - compute the dot product of two vectors x and y.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DDOT( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)

DOUBLE PRECISION FUNCTION DDOT_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION DOT( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y

REAL(8) FUNCTION DOT_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

double ddot(int n, double *x, int incx, double *y, int incy);

double ddot_64(long n, double *x, long incx, double *y, long incy);

PURPOSE

ddot compute the dot product of x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

ddoti - Compute the indexed dot product.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DDOTI(NZ, X, INDX, Y)

DOUBLE PRECISION X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

DOUBLE PRECISION FUNCTION DDOTI_64(NZ, X, INDX, Y)

DOUBLE PRECISION X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE DOUBLE PRECISION FUNCTION DOTI([NZ], X, INDX, Y)

```
REAL(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

DOUBLE PRECISION FUNCTION DOTI_64([NZ], X, INDX, Y)

REAL(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

DDOTI Compute the indexed dot product of a real sparse vector x stored in compressed form with a real vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + x(i) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

dezftb - computes a periodic sequence from its Fourier coefficients. DEZFTB is a simplified but slower version of DFFTB.
=head1 SYNOPSIS

```
SUBROUTINE DEZFTB( N, R, AZERO, A, B, WSAVE)
INTEGER N
DOUBLE PRECISION AZERO
DOUBLE PRECISION R(*), A(*), B(*), WSAVE(*)

SUBROUTINE DEZFTB_64( N, R, AZERO, A, B, WSAVE)
INTEGER*8 N
DOUBLE PRECISION AZERO
DOUBLE PRECISION R(*), A(*), B(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE DEZFTB( N, R, AZERO, A, B, WSAVE)
INTEGER :: N
REAL(8) :: AZERO
REAL(8), DIMENSION(:) :: R, A, B, WSAVE

SUBROUTINE DEZFTB_64( N, R, AZERO, A, B, WSAVE)
INTEGER(8) :: N
REAL(8) :: AZERO
REAL(8), DIMENSION(:) :: R, A, B, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dezftb(int n, double *r, double azero, double *a, double *b, double *wsave);

void dezftb_64(long n, double *r, double azero, double *a, double *b, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be synthesized. The method is most efficient when N is the product of small primes. N >= 0.
- **R (output)**
  On exit, the Fourier synthesis of the inputs.
- **AZERO (input)**
  On entry, the constant Fourier coefficient A0. Unchanged on exit.
- **A (input)**
  On entry, array that contains the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **B (input)**
  On entry, array that contains the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by DEZFTI.

NAME

dezftf - computes the Fourier coefficients of a periodic sequence. DEZFTF is a simplified but slower version of DFFTF.
=head1 SYNOPSIS

```
SUBROUTINE DEZFTF( N, R, AZERO, A, B, WSAVE)
INTEGER N
DOUBLE PRECISION AZERO
DOUBLE PRECISION R(*), A(*), B(*), WSAVE(*)

SUBROUTINE DEZFTF_64( N, R, AZERO, A, B, WSAVE)
INTEGER*8 N
DOUBLE PRECISION AZERO
DOUBLE PRECISION R(*), A(*), B(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE DEZFTF( N, R, AZERO, A, B, WSAVE)
INTEGER :: N
REAL(8) :: AZERO
REAL(8), DIMENSION(:) :: R, A, B, WSAVE

SUBROUTINE DEZFTF_64( N, R, AZERO, A, B, WSAVE)
INTEGER(8) :: N
REAL(8) :: AZERO
REAL(8), DIMENSION(:) :: R, A, B, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dezftf(int n, double *r, double azero, double *a, double *b, double *wsave);

void dezftf_64(long n, double *r, double azero, double *a, double *b, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is the product of small primes. N >= 0.
- **R (output)**
  On entry, a real array of length N containing the sequence to be transformed. On exit, R is unchanged.
- **AZERO (input)**
  On exit, the sum from i =1 to i =n of r(i)/n.
- **A (input)**
  On entry, array that contains the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **B (input)**
  On entry, array that contains the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by DEZFTI.

NAME

dezfti - initializes the array WSAVE, which is used in both DEZFTF and DEZFTB. =head1 SYNOPSIS

```
SUBROUTINE DEZFTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DEZFTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE DEZFTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE DEZFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dezfti(int n, double *wsave);

void dezfti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array with a dimension of at least (3 * N + 15). The same work array can be used for both DEZFTF and DEZFTB as long as N remains unchanged. Different WSAVE arrays are required for different values of N. This initialization does not have to be repeated between calls to DEZFTF or DEZFTB as long as N and WSAVE remain unchanged, thus subsequent transforms can be obtained faster than the first.

NAME

dfft2b - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFT2F followed by a call of DFFT2B will multiply the input sequence by M*N. =head1 SYNOPSIS

```
  SUBROUTINE DFFT2B( PLACE, M, N, A, LDA, B, LDB, WORK, LWORK)
  CHARACTER * 1 PLACE
  INTEGER M, N, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

  SUBROUTINE DFFT2B_64( PLACE, M, N, A, LDA, B, LDB, WORK, LWORK)
  CHARACTER * 1 PLACE
  INTEGER*8 M, N, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE FFT2B( PLACE, [M], [N], A, [LDA], B, [LDB], WORK, LWORK)
  CHARACTER(LEN=1) :: PLACE
  INTEGER :: M, N, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:) :: A, B

  SUBROUTINE FFT2B_64( PLACE, [M], [N], A, [LDA], B, [LDB], WORK,
 *       LWORK)
  CHARACTER(LEN=1) :: PLACE
  INTEGER(8) :: M, N, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dfft2b(char place, int m, int n, double *a, int lda, double *b, int ldb, double *work, int lwork);

void dfft2b_64(char place, long m, long n, double *a, long lda, double *b, long ldb, double *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Character. If PLACE = 'I' or 'i' (for in-place) , the input and output data are stored in array A. If PLACE = 'O' or 'o' (for out-of-place), the input data is stored in array B while the output is stored in A.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  Real array of dimension (LDA,N). On entry, the two-dimensional array A(LDA,N) contains the input data to be transformed if an in-place transform is requested. Otherwise, it is not referenced. Upon exit, results are stored in A(1:M,1:N).
- **LDA (input)**
  Integer specifying the leading dimension of A. If an out-of-place transform is desired LDA > = M. Else if an in-place transform is desired LDA > = 2*(M/2+1).
- **B (input/output)**
  Real array of dimension (2*LDB, N). On entry, if an out-of-place transform is requested B contains the input data. Otherwise, B is not referenced. B is unchanged upon exit.
- **LDB (input)**
  Integer. If an out-of-place transform is desired, 2*LDB is the leading dimension of the array B which contains the data to be transformed and 2*LDB > = 2*(M/2+1). Otherwise it is not referenced.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by DFFT2I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*N + MAX(M, 2*N) + 30)

NAME

dfft2f - compute the Fourier coefficients of a periodic sequence. The DFFT operations are unnormalized, so a call of DFFT2F followed by a call of DFFT2B will multiply the input sequence by M*N. =head1 SYNOPSIS

```
  SUBROUTINE DFFT2F( PLACE, FULL, M, N, A, LDA, B, LDB, WORK, LWORK)
  CHARACTER * 1 PLACE, FULL
  INTEGER M, N, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

  SUBROUTINE DFFT2F_64( PLACE, FULL, M, N, A, LDA, B, LDB, WORK,
 *      LWORK)
  CHARACTER * 1 PLACE, FULL
  INTEGER*8 M, N, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE FFT2F( PLACE, FULL, [M], [N], A, [LDA], B, [LDB], WORK,
 *       LWORK)
  CHARACTER(LEN=1) :: PLACE, FULL
  INTEGER :: M, N, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:) :: A, B

  SUBROUTINE FFT2F_64( PLACE, FULL, [M], [N], A, [LDA], B, [LDB],
 *       WORK, LWORK)
  CHARACTER(LEN=1) :: PLACE, FULL
  INTEGER(8) :: M, N, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dfft2f(char place, char full, int m, int n, double *a, int lda, double *b, int ldb, double *work, int lwork);

void dfft2f_64(char place, char full, long m, long n, double *a, long lda, double *b, long ldb, double *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Character. If PLACE = 'I' or 'i' (for in-place) , the input and output data are stored in array A. If PLACE = 'O' or 'o' (for out-of-place), the input data is stored in array B while the output is stored in A.
- **FULL (input)**
  Indicates whether or not to generate the full result matrix. 'F' or 'f' will cause DFFT2F to generate the full result matrix. Otherwise only a partial matrix that takes advantage of symmetry will be generated.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, a two-dimensional array `A(LDA,N)` that contains the data to be transformed. Upon exit, A is unchanged if an out-of-place transform is done. If an in-place transform with partial result is requested, `A(1:(M/2+1)*2,1:N)` will contain the transformed results. If an in-place transform with full result is requested, `A(1:2*M,1:N)` will contain complete transformed results.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA must be even if the transformed sequences are to be stored in A.

  If PLACE = ('O' or 'o') LDA > = M

  If PLACE = ('I' or 'i') LDA must be even. If

  FULL = ('F' or 'f'), LDA > = 2*M

  FULL is not ('F' or 'f'), LDA > = (M/2+1)*2
- **B (input/output)**
  Upon exit, a two-dimensional array `B(2*LDB,N)` that contains the transformed results if an out-of-place transform is done. Otherwise, B is not used.

  If an out-of-place transform is done and FULL is not 'F' or 'f', `B(1:(M/2+1)*2,1:N)` will contain the partial transformed results. If FULL = 'F' or 'f', `B(1:2*M,1:N)` will contain the complete transformed results.
- **LDB (input)**
  2*LDB is the leading dimension of the array B. If an in-place transform is desired LDB is ignored.

  If PLACE is ('O' or 'o') and

  FULL is ('F' or 'f'), LDB > = M

  FULL is not ('F' or 'f'), LDB > = M/2+1

  Note that even though LDB is used in the argument list, 2*LDB is the actual leading dimension of B.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by DFFT2I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*N + MAX(M, 2*N) + 30)

NAME

dfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

SYNOPSIS

```
SUBROUTINE DFFT2I( M, N, WORK)
INTEGER M, N
DOUBLE PRECISION WORK(*)

SUBROUTINE DFFT2I_64( M, N, WORK)
INTEGER*8 M, N
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2I( M, N, WORK)
INTEGER :: M, N
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT2I_64( M, N, WORK)
INTEGER(8) :: M, N
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void dfft2i(int m, int n, double *work);

void dfft2i_64(long m, long n, double *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (M + 2*N + MAX(M, 2*N) + 30) or greater. DFFT2I needs to be called only once to initialize array WORK before calling DFFT2F and/or DFFT2B if M, N and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dfft3b - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFT3F followed by a call of DFFT3B will multiply the input sequence by M*N*K. =head1 SYNOPSIS

```
  SUBROUTINE DFFT3B( PLACE, M, N, K, A, LDA, B, LDB, WORK, LWORK)
  CHARACTER * 1 PLACE
  INTEGER M, N, K, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,N,*), B(LDB,N,*), WORK(*)

  SUBROUTINE DFFT3B_64( PLACE, M, N, K, A, LDA, B, LDB, WORK, LWORK)
  CHARACTER * 1 PLACE
  INTEGER*8 M, N, K, LDA, LDB, LWORK
  DOUBLE PRECISION A(LDA,N,*), B(LDB,N,*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE FFT3B( PLACE, [M], [N], [K], A, [LDA], B, [LDB], WORK,
 *        LWORK)
  CHARACTER(LEN=1) :: PLACE
  INTEGER :: M, N, K, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:,:) :: A, B

  SUBROUTINE FFT3B_64( PLACE, [M], [N], [K], A, [LDA], B, [LDB], WORK,
 *        LWORK)
  CHARACTER(LEN=1) :: PLACE
  INTEGER(8) :: M, N, K, LDA, LDB, LWORK
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dfft3b(char place, int m, int n, int k, double *a, int lda, double *b, int ldb, double *work, int lwork);

void dfft3b_64(char place, long m, long n, long k, double *a, long lda, double *b, long ldb, double *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Select an in-place ('I' or 'i') or out-of-place ('O' or 'o') transform.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **K (input)**
  Integer specifying the number of planes to be transformed. It is most efficient when K is a product of small primes. K > = 0; when K = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, the three-dimensional array `A(LDA,N,K)` contains the data to be transformed if an in-place transform is requested. Otherwise, it is not referenced. Upon exit, results are stored in A(1:M,1:N,1:K).
- **LDA (input)**
  Integer specifying the leading dimension of A. If an out-of-place transform is desired LDA > = M. Else if an in-place transform is desired LDA > = 2*(M/2+1).
- **B (input/output)**
  Real array of dimension B(2*LDB,N,K). On entry, if an out-of-place transform is requested `B(1:2*(M/2+1),1:N,1:K)` contains the input data. Otherwise, B is not referenced. B is unchanged upon exit.
- **LDB (input)**
  If an out-of-place transform is desired, 2*LDB is the leading dimension of the array B which contains the data to be transformed and 2*LDB > = 2*(M/2+1). Otherwise it is not referenced.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by DFFT3I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*(N + K) + 4*K + 45).

NAME

dfft3f - compute the Fourier coefficients of a real periodic sequence. The DFFT operations are unnormalized, so a call of DFFT3F followed by a call of DFFT3B will multiply the input sequence by M*N*K. =head1 SYNOPSIS

```
 SUBROUTINE DFFT3F( PLACE, FULL, M, N, K, A, LDA, B, LDB, WORK,
*      LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER M, N, K, LDA, LDB, LWORK
 DOUBLE PRECISION A(LDA,N,*), B(LDB,N,*), WORK(*)

 SUBROUTINE DFFT3F_64( PLACE, FULL, M, N, K, A, LDA, B, LDB, WORK,
*      LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER*8 M, N, K, LDA, LDB, LWORK
 DOUBLE PRECISION A(LDA,N,*), B(LDB,N,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3F( PLACE, FULL, [M], [N], [K], A, [LDA], B, [LDB],
*       WORK, LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER :: M, N, K, LDA, LDB, LWORK
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:,:) :: A, B

 SUBROUTINE FFT3F_64( PLACE, FULL, [M], [N], [K], A, [LDA], B, [LDB],
*       WORK, LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER(8) :: M, N, K, LDA, LDB, LWORK
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dfft3f(char place, char full, int m, int n, int k, double *a, int lda, double *b, int ldb, double *work, int lwork);

void dfft3f_64(char place, char full, long m, long n, long k, double *a, long lda, double *b, long ldb, double *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Select an in-place ('I' or 'i') or out-of-place ('O' or 'o') transform.
- **FULL (input)**
  Select a full ('F' or 'f') or partial (' ') representation of the results. If the caller selects full representation then an MxNxK real array will transform to produce an MxNxK complex array. If the caller does not select full representation then an MxNxK real array will transform to a (M/2+1)xNxK complex array that takes advantage of the symmetry properties of a transformed real sequence.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **K (input)**
  Integer specifying the number of planes to be transformed. It is most efficient when K is a product of small primes. K > = 0; when K = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, a three-dimensional array `A(LDA,N,K)` that contains input data to be transformed. On exit, if an in-place transform is done and FULL is not 'F' or 'f', `A(1:2*(M/2+1),1:N,1:K)` will contain the partial transformed results. If FULL = 'F' or 'f', `A(1:2*M,1:N,1:K)` will contain the complete transformed results.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA must be even if the transformed sequences are to be stored in A.

  If PLACE = ('O' or 'o') LDA > = M

  If PLACE = ('I' or 'i') LDA must be even. If

  FULL = ('F' or 'f'), LDA > = 2*M

  FULL is not ('F' or 'f'), LDA > = 2*(M/2+1)
- **B (input/output)**
  Upon exit, a three-dimensional array `B(2*LDB,N,K)` that contains the transformed results if an out-of-place transform is done. Otherwise, B is not used.

  If an out-of-place transform is done and FULL is not 'F' or 'f', `B(1:2*(M/2+1),1:N,1:K)` will contain the partial transformed results. If FULL = 'F' or 'f', `B(1:2*M,1:N,1:K)` will contain the complete transformed results.
- **LDB (input)**
  2*LDB is the leading dimension of the array B. If an in-place transform is desired LDB is ignored.

  If PLACE is ('O' or 'o') and

  FULL is ('F' or 'f'), then LDB > = M

  FULL is not ('F' or 'f'), then LDB > = M/2 + 1

  Note that even though LDB is used in the argument list, 2*LDB is the actual leading dimension of B.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. WORK must have been initialized by DFFT3I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*(N + K) + 4*K + 45).

NAME

dfft3i - initialize the array WSAVE, which is used in both DFFT3F and DFFT3B.

SYNOPSIS

```
SUBROUTINE DFFT3I( M, N, K, WORK)
INTEGER M, N, K
DOUBLE PRECISION WORK(*)

SUBROUTINE DFFT3I_64( M, N, K, WORK)
INTEGER*8 M, N, K
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3I( M, N, K, WORK)
INTEGER :: M, N, K
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT3I_64( M, N, K, WORK)
INTEGER(8) :: M, N, K
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void dfft3i(int m, int n, int k, double *work);

void dfft3i_64(long m, long n, long k, double *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **K (input)**
  Number of planes to be transformed. K > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (M + 2*(N + K) + 30) or greater. DFFT3I needs to be called only once to initialize array WORK before calling DFFT3F and/or DFFT3B if M, N, K and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dfftb - compute a periodic sequence from its Fourier coefficients. The DFFT operations are unnormalized, so a call of DFFTF followed by a call of DFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE DFFTB( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DFFTB_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE FFTB_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dfftb(int n, double *x, double *wsave);

void dfftb_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (2 * N + 15) or greater and must have been initialized by DFFTI.

NAME

dfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of DFFTF followed by a call of DFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE DFFTF( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DFFTF_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE FFTF_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dfftf(int n, double *x, double *wsave);

void dfftf_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (2 * N + 15) or greater and must have been initialized by DFFTI.

NAME

dffti - initialize the array WSAVE, which is used in both DFFTF and DFFTB.

SYNOPSIS

```
SUBROUTINE DFFTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DFFTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE FFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dffti(int n, double *wsave);

void dffti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. DFFTI needs to be called only once to initialize array WORK before calling DFFTF and/or DFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dfftopt - compute the length of the closest fast FFT

SYNOPSIS

```
INTEGER FUNCTION DFFTOPT( LEN)
INTEGER LEN

INTEGER*8 FUNCTION DFFTOPT_64( LEN)
INTEGER*8 LEN
```

F95 INTERFACE

```
INTEGER FUNCTION DFFTOPT( LEN)
INTEGER :: LEN

INTEGER(8) FUNCTION DFFTOPT_64( LEN)
INTEGER(8) :: LEN
```

C INTERFACE

#include <sunperf.h>

int dfftopt(int len);

long dfftopt_64(long len);

PURPOSE

dfftopt computes the length of the closest fast FFT. Fast Fourier transform algorithms, including those used in Performance Library, work best with vector lengths that are products of small primes. For example, an FFT of length 32=2**5 will run faster than an FFT of prime length 31 because 32 is a product of small primes and 31 is not. If your application is such that you can taper or zero pad your vector to a larger length then this function may help you select a better length and run your FFT faster.

DFFTOPT will return an integer no smaller than the input argument N that is the closest number that is the product of small primes. DFFTOPT will return 16 for an input of N=16 and return 18=2*3*3 for an input of N=17.

Note that the length computed here is not guaranteed to be optimal, only to be a product of small primes. Also, the value returned may change as the underlying FFTs become capable of handling larger primes. For example, passing in N=51 to day will return 52=2*2*13 rather than 51=3*17 because the FFTs in Performance Li brary do not have fast radix 17 code. In the future, radix 17 code may be added and then N=51 will return 51.

ARGUMENTS

NAME

dfftz - initialize the trigonometric weight and factor tables or compute the forward Fast Fourier Transform of a double precision sequence. =head1 SYNOPSIS

```
 SUBROUTINE DFFTZ( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX Y(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(*), TRIGS(*), WORK(*)

 SUBROUTINE DFFTZ_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX Y(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX(8), DIMENSION(:) :: Y
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: X, TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK,
*       [LWORK], IERR)
 COMPLEX(8), DIMENSION(:) :: Y
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: X, TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void dfftz(int iopt, int n, double scale, double *x, doublecomplex *y, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void dfftz_64(long iopt, long n, double scale, double *x, doublecomplex *y, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

dfftz initializes the trigonometric weight and factor tables or computes the forward Fast Fourier Transform of a double precision sequence as follows: .Ve

```
                N-1
```

Y(k) = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

`k ranges from 0 to N-1`

i = `sqrt(-1)`

isign = -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

In real-to-complex transform of length N, the (N/2+1) complex output data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  On entry, X is a real array whose first N elements contain the sequence to be transformed.
- **Y (output)**
  Double complex array whose first (N/2+1) elements contain the transform results. X and Y may be the same array starting at the same memory location, in which case the dimension of X must be at least 2*(N/2+1). Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **TRIGS (input/output)**
  Double precision array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

dfftz2 - initialize the trigonometric weight and factor tables or compute the two-dimensional forward Fast Fourier Transform of a two-dimensional double precision array. =head1 SYNOPSIS

```
 SUBROUTINE DFFTZ2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX,*), TRIGS(*), WORK(*)

 SUBROUTINE DFFTZ2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: X

 SUBROUTINE FFT2_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void dfftz2(int iopt, int n1, int n2, double scale, double *x, int ldx, doublecomplex *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void dfftz2_64(long iopt, long n1, long n2, double scale, double *x, long ldx, doublecomplex *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

dfftz2 initializes the trigonometric weight and factor tables or computes the two-dimensional forward Fast Fourier Transform of a two-dimensional double precision array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the columns of the input array. One-dimensional FFTs are then computed along the rows of the intermediate results.
.Ve

```
                    N2-1  N1-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

In real-to-complex transform of length N1, the (N1/2+1) complex output data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small

primes. N1 > = 0. Unchanged on exit.

- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes N2 > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX, N2) that contains input data to be transformed. X and Y can be the same array.
- **LDX (input)**
  Leading dimension of X. LDX > = N1 if X is not the same array as Y. Else, LDX = 2*LDY. Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. LDY > = N1/2+1 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least MAX(N1, 2*N2) where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1) or (LDX not equal 2*LDY when X and Y are same array)

  -5 = (LDY < N1/2+1)

  -6 = (LWORK not equal 0) and (LWORK < `MAX(N1,2*N2))`

  -7 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output array Y(1:LDY, 1:N2) is overwritten.

NAME

dfftz3 - initialize the trigonometric weight and factor tables or compute the three-dimensional forward Fast Fourier Transform of a three-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE DFFTZ3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY1,LDY2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX1,LDX2,*), TRIGS(*), WORK(*)

 SUBROUTINE DFFTZ3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY1,LDY2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX1,LDX2,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: Y
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:,:) :: X

 SUBROUTINE FFT3_64( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1],
*       LDX2, Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: Y
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void dfftz3(int iopt, int n1, int n2, int n3, double scale, double *x, int ldx1, int ldx2, doublecomplex *y, int ldy1, int ldy2, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void dfftz3_64(long iopt, long n1, long n2, long n3, double scale, double *x, long ldx1, long ldx2, doublecomplex *y, long ldy1, long ldy2, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

dfftz3 initializes the trigonometric weight and factor tables or computes the three-dimensional forward Fast Fourier Transform of a three-dimensional double complex array. .Ve

```
                        N3-1  N2-1  N1-1
```

`Y(k1,k2,k3)` = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                        j3=0  j2=0  j1=0
```

`.Ve`

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double precision array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed. X can be same array as Y.
- **LDX1 (input)**
  first dimension of X. If X is not same array as Y, LDX1 > = N1 Else, LDX1 = 2*LDY1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. LDY1 > = N1/2+1 Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least (MAX(N,2*N2,2*N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1) or (LDX not equal 2*LDY when X and Y are same array)

  -6 = (LDX2 < N2)

  -7 = (LDY1 < N1/2+1)

  -8 = (LDY2 < N2) or (LDY2 not equal LDX2 when X and Y are same array)

  -9 = (LWORK not equal 0) and (LWORK < (MAX(N,2*N2,2*N3) + 16*N3))

  -10 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output subarray Y(1:LDY1, 1:N2, 1:N3) is overwritten.

NAME

dfftzm - initialize the trigonometric weight and factor tables or compute the one-dimensional forward Fast Fourier Transform of a set of double precision data sequences stored in a two-dimensional array. =head1 SYNOPSIS

```
 SUBROUTINE DFFTZM( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY,*)
 INTEGER IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX,*), TRIGS(*), WORK(*)

 SUBROUTINE DFFTZM_64( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX Y(LDY,*)
 INTEGER*8 IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION X(LDX,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, [M], [N], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: Y
 INTEGER :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: X

 SUBROUTINE FFTM_64( IOPT, [M], [N], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: Y
 INTEGER(8) :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void dfftzm(int iopt, int m, int n, double scale, double *x, int ldx, doublecomplex *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void dfftzm_64(long iopt, long m, long n, double scale, double *x, long ldx, doublecomplex *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

dfftzm initializes the trigonometric weight and factor tables or computes the one-dimensional forward Fast Fourier Transform of a set of double precision data sequences stored in a two-dimensional array: .Ve

```
                    N1-1
```

Y(k,l) = scale * SUM W*X(j,l)

```
                    j=0
```

.Ve

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N1)`

In real-to-complex transform of length N1, the (N1/2+1) complex output data points stored are the positive-frequency half of the spectrum of the discrete Fourier transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **M (input)**
  Integer specifying length of the input sequences. M is most efficient when it is a product of small primes. M > = 0. Unchanged on exit.
- **N (input)**
  Integer specifying number of input sequences. N > = 0. Unchanged on exit.
- **SCALE (input)**

Double precision scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  X is a double precision array of dimensions (LDX, N) that contains the sequences to be transformed stored in its columns.
- **LDX (input)**
  Leading dimension of X. If X and Y are the same array, LDX = 2*LDY Else LDX > = M Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY, N) that contains the transform results of the input sequences. X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. LDY > = M/2 + 1 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*M that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of M. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least M. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = M < 0

  -3 = N < 0

  -4 = (LDX < M) or (LDX not equal 2*LDY when X and Y are same array)

  -4 = (LDY < M/2 + 1)

  -6 = (LWORK not equal 0) and (LWORK < M)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

dgbbrd - reduce a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation

---

SYNOPSIS

```
 SUBROUTINE DGBBRD( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q, LDQ,
*      PT, LDPT, C, LDC, WORK, INFO)
 CHARACTER * 1 VECT
 INTEGER M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 DOUBLE PRECISION AB(LDAB,*), D(*), E(*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)

 SUBROUTINE DGBBRD_64( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q,
*      LDQ, PT, LDPT, C, LDC, WORK, INFO)
 CHARACTER * 1 VECT
 INTEGER*8 M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 DOUBLE PRECISION AB(LDAB,*), D(*), E(*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBBRD( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D, E,
*       Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 INTEGER :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: AB, Q, PT, C

 SUBROUTINE GBBRD_64( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D,
*       E, Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 INTEGER(8) :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: AB, Q, PT, C
```

C INTERFACE

#include <sunperf.h>

void dgbbrd(char vect, int m, int n, int ncc, int kl, int ku, double *ab, int ldab, double *d, double *e, double *q, int ldq, double *pt, int ldpt, double *c, int ldc, int *info);

void dgbbrd_64(char vect, long m, long n, long ncc, long kl, long ku, double *ab, long ldab, double *d, double *e, double *q, long ldq, double *pt, long ldpt, double *c, long ldc, long *info);

PURPOSE

dgbbrd reduces a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation: Q' * A * P = B.

The routine computes B, and optionally forms Q or P', or computes Q'*C for a given matrix C.

ARGUMENTS

- **VECT (input)**
  Specifies whether or not the matrices Q and P' are to be formed. = 'N': do not form Q or P';

  ```
  = 'Q': form Q only;

  = 'P': form P' only;

  = 'B': form both.
  ```
- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC >= 0.
- **KL (input)**
  The number of subdiagonals of the matrix A. KL >= 0.
- **KU (input)**
  The number of superdiagonals of the matrix A. KU >= 0.
- **AB (input/output)**
  On entry, the m-by-n band matrix A, stored in rows 1 to KL+KU+1. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl). On exit, A is overwritten by values generated during the reduction.
- **LDAB (input)**
  The leading dimension of the array A. LDAB >= KL+KU+1.
- **D (output)**
  The diagonal elements of the bidiagonal matrix B.
- **E (output)**

The superdiagonal elements of the bidiagonal matrix B.

- **Q (output)**
  If VECT = 'Q' or 'B', the m-by-m orthogonal matrix Q. If VECT = 'N' or 'P', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,M)` if VECT = 'Q' or 'B'; LDQ >= 1 otherwise.
- **PT (output)**
  If VECT = 'P' or 'B', the n-by-n orthogonal matrix P'. If VECT = 'N' or 'Q', the array PT is not referenced.
- **LDPT (input)**
  The leading dimension of the array PT. LDPT >= `max(1,N)` if VECT = 'P' or 'B'; LDPT >= 1 otherwise.
- **C (input/output)**
  On entry, an m-by-ncc matrix C. On exit, C is overwritten by Q'*C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)` if NCC > 0; LDC >= 1 if NCC = 0.
- **WORK (workspace)**
  `dimension(MAX(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

dgbcon - estimate the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm,

---

SYNOPSIS

```
 SUBROUTINE DGBCON( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, NSUB, NSUPER, LDA, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DGBCON_64( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, NSUB, NSUPER, LDA, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBCON( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT, ANORM,
*        RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, NSUB, NSUPER, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GBCON_64( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT,
*        ANORM, RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, NSUB, NSUPER, LDA, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
```

```
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgbcon(char norm, int n, int nsub, int nsuper, double *a, int lda, int *ipivot, double anorm, double *rcond, int *info);

void dgbcon_64(char norm, long n, long nsub, long nsuper, double *a, long lda, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

dgbcon estimates the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**

The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  dimension (N)
- **INFO (output)**

  `= 0:  successful exit`

  `< 0: if INFO  = -i, the i-th argument had an illegal value`

NAME

dgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE DGBEQU( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC, ROWCN,
*       COLCN, AMAX, INFO)
 INTEGER M, N, NSUB, NSUPER, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION A(LDA,*), ROWSC(*), COLSC(*)

 SUBROUTINE DGBEQU_64( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC,
*       ROWCN, COLCN, AMAX, INFO)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION A(LDA,*), ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GBEQU( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCN, COLCN, AMAX, [INFO])
 INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GBEQU_64( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCN, COLCN, AMAX, [INFO])
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgbequ(int m, int n, int nsub, int nsuper, double *a, int lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, int *info);

void dgbequ_64(long m, long n, long nsub, long nsuper, double *a, long lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, long *info);

PURPOSE

dgbequ computes row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input)**
  The band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **ROWSC (output)**
  If INFO = 0, or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, and i is

  < = M: the i-th row of A is exactly zero

  > M: the (i-M)-th column of A is exactly zero

NAME

dgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE DGBMV( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 INTEGER M, N, NSUB, NSUPER, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)

 SUBROUTINE DGBMV_64( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE GBMV( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GBMV_64( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgbmv(char transa, int m, int n, int nsub, int nsuper, double alpha, double *a, int lda, double *x, int incx, double beta, double *y, int incy);

void dgbmv_64(char transa, long m, long n, long nsub, long nsuper, double alpha, double *a, long lda, double *x, long incx, double beta, double *y, long incy);

PURPOSE

dgbmv performs one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with nsub sub-diagonals and nsuper super-diagonals.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*A'*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **NSUB (input)**
  On entry, NSUB specifies the number of sub-diagonals of the matrix A. NSUB > = 0. Unchanged on exit.
- **NSUPER (input)**
  On entry, NSUPER specifies the number of super-diagonals of the matrix A. NSUPER > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading (nsub + nsuper + 1) by n part of the array A must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (nsuper + 1) of the array, the first super-diagonal starting at position 2 in row nsuper, the first sub-diagonal starting at position 1 in row (nsuper + 2), and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left nsuper by nsuper triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
  K = NSUPER + 1 - J
  DO 10, I = MAX( 1, J - NSUPER ), MIN( M, J  + NSUB )
    A( K + I, J ) = matrix( I, J )
```

```
10   CONTINUE
20 CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (nsub + nsuper + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DGBRFS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF, LDAF,
*       IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DGBRFS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*       LDAF, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBRFS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GBRFS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dgbrfs(char transa, int n, int nsub, int nsuper, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dgbrfs_64(char transa, long n, long nsub, long nsuper, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dgbrfs improves the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` < =i < =min(n,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.
- **AF (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB*NSUPER+1.
- **IPIVOT (input)**
  The pivot indices from SGBTRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).

- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGBTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dgbsv - compute the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE DGBSV( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*       INFO)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DGBSV_64( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*       INFO)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GBSV( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*        [LDB], [INFO])
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GBSV_64( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*        [LDB], [INFO])
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgbsv(int n, int nsub, int nsuper, int nrhs, double *a, int lda, int *ipivot, double *b, int ldb, int *info);

void dgbsv_64(long n, long nsub, long nsuper, long nrhs, double *a, long lda, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dgbsv computes the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = L * U, where L is a product of permutation and unit lower triangular matrices with KL subdiagonals, and U is upper triangular with KL+KU superdiagonals. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUB+NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+NSUB) On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

```
 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +       *    *    *   u14  u25  u36
 *    *    +    +    +    +       *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *      m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *      m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

dgbsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE DGBSVX( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*      LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DGBSVX_64( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA,
*      AF, LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND,
*      FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSVX( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GBSVX_64( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A,
*       [LDA], AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dgbsvx(char fact, char transa, int n, int nsub, int nsuper, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, char equed, double *rowsc, double *colsc, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dgbsvx_64(char fact, char transa, long n, long nsub, long nsuper, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, char equed, double *rowsc, double *colsc, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dgbsvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed by this subroutine:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = L * U,

where L is a product of permutation and unit lower triangular
matrices with KL subdiagonals, and U is upper triangular with
KL+KU superdiagonals.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```
- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Transpose)
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the matrix A in band storage, in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: A(NSUPER+1+i-j,j) = A(i,j) for max(1,j-NSUPER) < =i < =min(N,j+kl)

  If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = diag(ROWSC) * A

  EQUED = 'COLSC': A : = A * diag(COLSC)

  EQUED = 'B': A : = diag(ROWSC) * A * diag(COLSC).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns details of the LU factorization of A.

  If FACT = 'E', then AF is an output argument and on exit returns details of the LU factorization of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB+NSUPER+1.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = L*U as computed by SGBTRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the equilibrated matrix A.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)` On exit, `WORK(1)` contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If `WORK(1)` is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then `WORK(1)` contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

dgbtf2 - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE DGBTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO)
INTEGER M, N, KL, KU, LDAB, INFO
INTEGER IPIV(*)
DOUBLE PRECISION AB(LDAB,*)

SUBROUTINE DGBTF2_64( M, N, KL, KU, AB, LDAB, IPIV, INFO)
INTEGER*8 M, N, KL, KU, LDAB, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE GBTF2( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
INTEGER :: M, N, KL, KU, LDAB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: AB

SUBROUTINE GBTF2_64( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
INTEGER(8) :: M, N, KL, KU, LDAB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void dgbtf2(int m, int n, int kl, int ku, double *ab, int ldab, int *ipiv, int *info);

void dgbtf2_64(long m, long n, long kl, long ku, double *ab, long ldab, long *ipiv, long *info);

PURPOSE

dgbtf2 computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **KL (input)**
  The number of subdiagonals within the band of A. KL >= 0.
- **KU (input)**
  The number of superdiagonals within the band of A. KU >= 0.
- **AB (input/output)**
  On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1. See below for further details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= 2*KL+KU+1.
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, KL = 2, KU = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U, because of fill-in resulting from the row

interchanges.

NAME

dgbtrf - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE DGBTRF( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
INTEGER M, N, NSUB, NSUPER, LDA, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DGBTRF_64( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GBTRF( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GBTRF_64( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgbtrf(int m, int n, int nsub, int nsuper, double *a, int lda, int *ipivot, int *info);

void dgbtrf_64(long m, long n, long nsub, long nsuper, double *a, long lda, long *ipivot, long *info);

PURPOSE

dgbtrf computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

dgbtrs - solve a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF

SYNOPSIS

```
 SUBROUTINE DGBTRS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DGBTRS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GBTRS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GBTRS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgbtrs(char transa, int n, int nsub, int nsuper, int nrhs, double *a, int lda, int *ipivot, double *b, int ldb, int *info);

void dgbtrs_64(char transa, long n, long nsub, long nsuper, long nrhs, double *a, long lda, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dgbtrs solves a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

< 0: if INFO = -i, the i-th argument had an illegal value

NAME

dgebak - form the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL

SYNOPSIS

```
 SUBROUTINE DGEBAK( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION SCALE(*), V(LDV,*)

 SUBROUTINE DGEBAK_64( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV,
*      INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION SCALE(*), V(LDV,*)
```

F95 INTERFACE

```
 SUBROUTINE GEBAK( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: SCALE
 REAL(8), DIMENSION(:,:) :: V

 SUBROUTINE GEBAK_64( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: SCALE
 REAL(8), DIMENSION(:,:) :: V
```

C INTERFACE

#include <sunperf.h>

void dgebak(char job, char side, int n, int ilo, int ihi, double *scale, int m, double *v, int ldv, int *info);

void dgebak_64(char job, char side, long n, long ilo, long ihi, double *scale, long m, double *v, long ldv, long *info);

PURPOSE

dgebak forms the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required: = 'N', do nothing, return immediately; = 'P', do backward transformation for permutation only; = 'S', do backward transformation for scaling only; = 'B', do backward transformations for both permutation and scaling. JOB must be the same as the argument JOB supplied to SGEBAL.
- **SIDE (input)**

  ```
  = 'R':  V contains right eigenvectors;

  = 'L':  V contains left eigenvectors.
  ```
- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by SGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description for ILO.
- **SCALE (input)**
  Details of the permutation and scaling factors, as returned by SGEBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**
  On entry, the matrix of right or left eigenvectors to be transformed, as returned by SHSEIN or STREVC. On exit, V is overwritten by the transformed eigenvectors.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

dgebal - balance a general real matrix A

SYNOPSIS

```
SUBROUTINE DGEBAL( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
INTEGER N, LDA, ILO, IHI, INFO
DOUBLE PRECISION A(LDA,*), SCALE(*)

SUBROUTINE DGEBAL_64( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
INTEGER*8 N, LDA, ILO, IHI, INFO
DOUBLE PRECISION A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE GEBAL( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER :: N, LDA, ILO, IHI, INFO
REAL(8), DIMENSION(:) :: SCALE
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GEBAL_64( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER(8) :: N, LDA, ILO, IHI, INFO
REAL(8), DIMENSION(:) :: SCALE
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgebal(char job, int n, double *a, int lda, int *ilo, int *ihi, double *scale, int *info);

void dgebal_64(char job, long n, double *a, long lda, long *ilo, long *ihi, double *scale, long *info);

PURPOSE

dgebal balances a general real matrix A. This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to ILO-1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, SCALE(I)  = 1.0
for i  = 1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description for ILO.
- **SCALE (output)**
  Details of the permutations and scaling factors applied to A. If `P(j)` is the index of the row and column interchanged with row and column j and `D(j)` is the scaling factor applied to row and column j, then `SCALE(j)` = `P(j)` for j = 1,...,ILO-1 = `D(j)` for j = ILO,...,IHI = `P(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The permutations consist of row and column interchanges which put the matrix in the form

```
           ( T1   X   Y  )
P A P  = (  0   B   Z  )
           (  0   0   T2 )
```

where T1 and T2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ILO and IHI mark the starting and ending columns of the submatrix B. Balancing consists of applying a diagonal similarity transformation `inv(D)` * B * D to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

```
( T1     X*D          Y    )
(  0  inv(D)*B*D  inv(D)*Z ).
(  0      0           T2   )
```

Information about the permutations P and the diagonal matrix D is returned in the vector SCALE.

This subroutine is based on the EISPACK routine BALANC.

Modified by Tzu-Yi Chen, Computer Science Division, University of California at Berkeley, USA

NAME

dgebrd - reduce a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation

---

SYNOPSIS

```
 SUBROUTINE DGEBRD( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 INTEGER M, N, LDA, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), D(*), E(*), TAUQ(*), TAUP(*), WORK(*)

 SUBROUTINE DGEBRD_64( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 INTEGER*8 M, N, LDA, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), D(*), E(*), TAUQ(*), TAUP(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBRD( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 INTEGER :: M, N, LDA, LWORK, INFO
 REAL(8), DIMENSION(:) :: D, E, TAUQ, TAUP, WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GEBRD_64( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 REAL(8), DIMENSION(:) :: D, E, TAUQ, TAUP, WORK
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgebrd(int m, int n, double *a, int lda, double *d, double *e, double *tauq, double *taup, int *info);

void dgebrd_64(long m, long n, double *a, long lda, double *d, double *e, double *tauq, double *taup, long *info);

PURPOSE

dgebrd reduces a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation: Q**T * A * P = B.

If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.

ARGUMENTS

- **M (input)**
  The number of rows in the matrix A. M > = 0.
- **N (input)**
  The number of columns in the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N general matrix to be reduced. On exit, if m > = n, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors; if m < n, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **D (output)**
  The diagonal elements of the bidiagonal matrix B: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the bidiagonal matrix B: if m > = n, `E(i)` = `A(i,i+1)` for i = 1,2,...,n-1; if m < n, `E(i)` = `A(i+1,i)` for i = 1,2,...,m-1.
- **TAUQ (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.
- **TAUP (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,M,N). For optimum performance LWORK > = (M+N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrices Q and P are represented as products of elementary reflectors:

If m >= n,

```
Q  = H(1) H(2) . . . H(n)  and  P  = G(1) G(2) . . . G(n-1)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are real scalars, and v and u are real vectors; `v(1:i-1)` = 0, `v(i)` = 1, and `v(i+1:m)` is stored on exit in A(i+1:m,i); `u(1:i)` = 0, `u(i+1)` = 1, and `u(i+2:n)` is stored on exit in A(i,i+2:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

If m < n,

```
Q  = H(1) H(2) . . . H(m-1)  and  P  = G(1) G(2) . . . G(m)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are real scalars, and v and u are real vectors; `v(1:i)` = 0, `v(i+1)` = 1, and `v(i+2:m)` is stored on exit in A(i+2:m,i); `u(1:i-1)` = 0, `u(i)` = 1, and `u(i+1:n)` is stored on exit in A(i,i+1:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

The contents of A on exit are illustrated by the following examples:

m = 6 and n = 5 (m > n): m = 5 and n = 6 (m < n):

```
(  d   e   u1  u1  u1 )          (  d   u1  u1  u1  u1  u1 )
(  v1  d   e   u2  u2 )          (  e   d   u2  u2  u2  u2 )
(  v1  v2  d   e   u3 )          (  v1  e   d   u3  u3  u3 )
(  v1  v2  v3  d   e  )          (  v1  v2  e   d   u4  u4 )
(  v1  v2  v3  v4  d  )          (  v1  v2  v3  e   d   u5 )
(  v1  v2  v3  v4  v5 )
```

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining H(i), and ui an element of the vector defining G(i).

NAME

dgecon - estimate the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF

SYNOPSIS

```
 SUBROUTINE DGECON( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, LDA, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DGECON_64( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, LDA, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GECON( NORM, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GECON_64( NORM, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8) :: ANORM, RCOND
```

```
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgecon(char norm, int n, double *a, int lda, double anorm, double *rcond, int *info);

void dgecon_64(char norm, long n, double *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

dgecon estimates the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(4*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE DGEEQU( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN, AMAX,
*      INFO)
 INTEGER M, N, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION A(LDA,*), ROWSC(*), COLSC(*)

 SUBROUTINE DGEEQU_64( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN,
*      AMAX, INFO)
 INTEGER*8 M, N, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION A(LDA,*), ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEQU( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*       AMAX, [INFO])
 INTEGER :: M, N, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GEEQU_64( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*       AMAX, [INFO])
 INTEGER(8) :: M, N, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgeequ(int m, int n, double *a, int lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, int *info);

void dgeequ_64(long m, long n, double *a, long lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, long *info);

PURPOSE

dgeequ computes row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input)**
  The M-by-N matrix whose equilibration factors are to be computed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **ROWSC (output)**
  If INFO = 0 or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: if INFO = i, and i is

< = M: the i-th row of A is exactly zero

> M: the (i-M)-th column of A is exactly zero

NAME

dgees - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE DGEES( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, WR, WI, Z,
*      LDZ, WORK, LDWORK, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL WORK3(*)
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DGEES_64( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, WR, WI,
*      Z, LDZ, WORK, LDWORK, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 WORK3(*)
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEES( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, WR, WI,
*       Z, [LDZ], [WORK], [LDWORK], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: WORK3
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: A, Z

 SUBROUTINE GEES_64( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, WR,
*       WI, Z, [LDZ], [WORK], [LDWORK], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL(8) :: SELECT
```

```
LOGICAL(8), DIMENSION(:) :: WORK3
REAL(8), DIMENSION(:) :: WR, WI, WORK
REAL(8), DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void dgees(char jobz, char sortev, logical(*select)(double,double), int n, double *a, int lda, int *nout, double *wr, double *wi, double *z, int ldz, int *info);

void dgees_64(char jobz, char sortev, logical(*select)(double,double), long n, double *a, long lda, long *nout, double *wr, double *wi, double *z, long ldz, long *info);

PURPOSE

dgees computes for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**T).

Optionally, it also orders the eigenvalues on the diagonal of the real Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A matrix is in real Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

```
[  a  b  ]

[  c  a  ]
```

where b*c < 0. The eigenvalues of such a block are a +- sqrt(bc).

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.
  ```

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

  ```
  = 'S': Eigenvalues are ordered (see SELECT).
  ```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to sort to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `WR(j)+sqrt(-1)*WI(j)` is selected if `SELECT(WR(j),WI(j))` is true; i.e., if either one of a complex

conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy SELECT(WR(j),WI(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case INFO is set to N+2 (see INFO below).

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten by its real Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues (after sorting) for which SELECT is true. (Complex conjugate pairs for which SELECT is true for either eigenvalue count as 2.)
- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues in the same order that they appear on the diagonal of the output Schur form T. Complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **Z (output)**
  If JOBZ = 'V', Z contains the orthogonal matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1; if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) contains the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,3*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK3 (workspace)**
  dimension(N) Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N: the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of WR and WI contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the matrix which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

dgeesx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE DGEESX( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT, WR,
*      WI, Z, LDZ, SRCONE, RCONV, WORK, LDWORK, IWORK2, LDWRK2, BWORK3,
*      INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 INTEGER N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER IWORK2(*)
 LOGICAL SELECT
 LOGICAL BWORK3(*)
 DOUBLE PRECISION SRCONE, RCONV
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DGEESX_64( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT,
*      WR, WI, Z, LDZ, SRCONE, RCONV, WORK, LDWORK, IWORK2, LDWRK2,
*      BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER*8 IWORK2(*)
 LOGICAL*8 SELECT
 LOGICAL*8 BWORK3(*)
 DOUBLE PRECISION SRCONE, RCONV
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEESX( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA], NOUT,
*       WR, WI, Z, [LDZ], SRCONE, RCONV, [WORK], [LDWORK], [IWORK2],
*       [LDWRK2], [BWORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER, DIMENSION(:) :: IWORK2
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: BWORK3
```

```
  REAL(8) :: SRCONE, RCONV
  REAL(8), DIMENSION(:) :: WR, WI, WORK
  REAL(8), DIMENSION(:,:) :: A, Z

  SUBROUTINE GEESX_64( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA],
*       NOUT, WR, WI, Z, [LDZ], SRCONE, RCONV, [WORK], [LDWORK], [IWORK2],
*       [LDWRK2], [BWORK3], [INFO])
  CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
  INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
  INTEGER(8), DIMENSION(:) :: IWORK2
  LOGICAL(8) :: SELECT
  LOGICAL(8), DIMENSION(:) :: BWORK3
  REAL(8) :: SRCONE, RCONV
  REAL(8), DIMENSION(:) :: WR, WI, WORK
  REAL(8), DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void dgeesx(char jobz, char sortev, logical(*select)(double,double), char sense, int n, double *a, int lda, int *nout, double *wr, double *wi, double *z, int ldz, double *srcone, double *rconv, int *info);

void dgeesx_64(char jobz, char sortev, logical(*select)(double,double), char sense, long n, double *a, long lda, long *nout, double *wr, double *wi, double *z, long ldz, double *srcone, double *rconv, long *info);

PURPOSE

dgeesx computes for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**T).

Optionally, it also orders the eigenvalues on the diagonal of the real Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (RCONDV). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see Section 4.10 of the LAPACK Users' Guide (where these quantities are called s and sep respectively).

A real matrix is in real Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

```
          [  a  b  ]

          [  c  a  ]
```

where b*c < 0. The eigenvalues of such a block are a +- sqrt(bc).

ARGUMENTS

- **JOBZ (input)**

```
= 'N': Schur vectors are not computed;

= 'V': Schur vectors are computed.
```

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S': Eigenvalues are ordered (see SELECT).
```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to sort to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `WR(j)+sqrt(-1)*WI(j)` is selected if `SELECT(WR(j),WI(j))` is true; i.e., if either one of a complex conjugate pair of eigenvalues is selected, then both are. Note that a selected complex eigenvalue may no longer satisfy `SELECT(WR(j),WI(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case INFO may be set to N+3 (see INFO below).
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
= 'E': Computed for average of selected eigenvalues only;

= 'V': Computed for selected right invariant subspace only;

= 'B': Computed for both.
If SENSE  = 'E', 'V' or 'B', SORTEV must equal 'S'.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A is overwritten by its real Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues (after sorting) for which SELECT is true. (Complex conjugate pairs for which SELECT is true for either eigenvalue count as 2.)
- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **Z (output)**
  If JOBZ = 'V', Z contains the orthogonal matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.
- **SRCONE (output)**
  If SENSE = 'E' or 'B', SRCONE contains the reciprocal condition number for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONV (output)**

If SENSE = 'V' or 'B', RCONV contains the reciprocal condition number for the selected right invariant subspace. Not referenced if SENSE = 'N' or 'E'.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,3*N). Also, if SENSE = 'E' or 'V' or 'B', LDWORK > = N+2*NOUT*(N-NOUT), where NOUT is the number of selected eigenvalues computed by this routine. Note that N+2*NOUT*(N-NOUT) < = N+N*N/2. For good performance, LDWORK must generally be larger.
- **IWORK2 (workspace)**
  Not referenced if SENSE = 'N' or 'E'. On exit, if INFO = 0, `IWORK2(1)` returns the optimal LDWRK2.
- **LDWRK2 (input)**
  The dimension of the array IWORK2. LDWRK2 > = 1; if SENSE = 'V' or 'B', LDWRK2 > = NOUT*(N-NOUT).
- **BWORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N: the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of WR and WI contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the transformation which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

dgeev - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE DGEEV( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR,
*      LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER N, LDA, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE DGEEV_64( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR,
*      LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER*8 N, LDA, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEV( JOBVL, JOBVR, [N], A, [LDA], WR, WI, VL, [LDVL],
*       VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: A, VL, VR

 SUBROUTINE GEEV_64( JOBVL, JOBVR, [N], A, [LDA], WR, WI, VL, [LDVL],
*       VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER(8) :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: A, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dgeev(char jobvl, char jobvr, int n, double *a, int lda, double *wr, double *wi, double *vl, int ldvl, double *vr, int ldvr, int *info);

void dgeev_64(char jobvl, char jobvr, long n, double *a, long lda, double *wr, double *wi, double *vl, long ldvl, double *vr, long ldvr, long *info);

PURPOSE

dgeev computes for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N': left eigenvectors of A are not computed;

  = 'V': left eigenvectors of A are computed.
  ```
- **JOBVR (input)**

  ```
  = 'N': right eigenvectors of A are not computed;

  = 'V': right eigenvectors of A are computed.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)` + i*VL(:,j+1) and

  `u(j+1)` = `VL(:,j)` - i*VL(:,j+1).
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)` + i*VR(:,j+1) and

  `v(j+1)` = `VR(:,j)` - i*VR(:,j+1).
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,3*N), and if JOBVL = 'V' or JOBVR = 'V', LDWORK >= 4*N. For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors have been computed;
elements i+1:N of WR and WI contain eigenvalues which
have converged.
```

NAME

dgeevx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE DGEEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, WR, WI,
*      VL, LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, IWORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 INTEGER N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER IWORK2(*)
 DOUBLE PRECISION ABNRM
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), SCALE(*), RCONE(*), RCONV(*), WORK(*)

 SUBROUTINE DGEEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, WR,
*      WI, VL, LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*      WORK, LDWORK, IWORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 INTEGER*8 N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER*8 IWORK2(*)
 DOUBLE PRECISION ABNRM
 DOUBLE PRECISION A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), SCALE(*), RCONE(*), RCONV(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], WR,
*       WI, VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*       WORK, [LDWORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 INTEGER :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK2
 REAL(8) :: ABNRM
 REAL(8), DIMENSION(:) :: WR, WI, SCALE, RCONE, RCONV, WORK
 REAL(8), DIMENSION(:,:) :: A, VL, VR

 SUBROUTINE GEEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], WR,
*       WI, VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*       WORK, [LDWORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 INTEGER(8) :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2
 REAL(8) :: ABNRM
 REAL(8), DIMENSION(:) :: WR, WI, SCALE, RCONE, RCONV, WORK
 REAL(8), DIMENSION(:,:) :: A, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dgeevx(char balanc, char jobvl, char jobvr, char sense, int n, double *a, int lda, double *wr, double *wi, double *vl, int ldvl, double *vr, int ldvr, int

*ilo, int *ihi, double *scale, double *abnrm, double *rcone, double *rconv, double *work, int ldwork, int *info);

void dgeevx_64(char balanc, char jobvl, char jobvr, char sense, long n, double *a, long lda, double *wr, double *wi, double *vl, long ldvl, double *vr, long ldvr, long *ilo, long *ihi, double *scale, double *abnrm, double *rcone, double *rconv, double *work, long ldwork, long *info);

PURPOSE

dgeevx computes for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, SCALE, and ABNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right

eigenvectors (RCONDV).

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D**(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide.

ARGUMENTS

- **BALANC (input)**
  Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. = 'N': Do not diagonally scale or permute;

  ```
   = 'P': Perform permutations to make the matrix more nearly
  upper triangular. Do not diagonally scale;
   = 'S': Diagonally scale the matrix, i.e. replace A by
  D*A*D**(-1), where D is a diagonal matrix chosen
  to make the rows and columns of A more equal in
  norm. Do not permute;
   = 'B': Both diagonally scale and permute A.
  ```

  Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
- **JOBVL (input)**

  ```
   = 'N': left eigenvectors of A are not computed;

   = 'V': left eigenvectors of A are computed.
  If SENSE  = 'E' or 'B', JOBVL must  = 'V'.
  ```
- **JOBVR (input)**

  ```
   = 'N': right eigenvectors of A are not computed;

   = 'V': right eigenvectors of A are computed.
  If SENSE  = 'E' or 'B', JOBVR must  = 'V'.
  ```
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

= 'E': Computed for eigenvalues only;

= 'V': Computed for right eigenvectors only;

= 'B': Computed for eigenvalues and right eigenvectors.

If SENSE = 'E' or 'B', both left and right eigenvectors must also be computed (JOBVL = 'V' and JOBVR = 'V').

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten. If JOBVL = 'V' or JOBVR = 'V', A contains the real Schur form of the balanced version of the input matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors u(j) are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. If the j-th eigenvalue is real, then u(j) = VL(:,j), the j-th column of VL. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then u(j) = VL(:,j) + i*VL(:,j+1) and

  u(j+1) = VL(:,j) - i*VL(:,j+1).
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors v(j) are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. If the j-th eigenvalue is real, then v(j) = VR(:,j), the j-th column of VR. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then v(j) = VR(:,j) + i*VR(:,j+1) and

  v(j+1) = VR(:,j) - i*VR(:,j+1).
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced A(i,j) = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **IHI (output)**
  See the description of ILO.
- **SCALE (output)**
  Details of the permutations and scaling factors applied when balancing A. If P(j) is the index of the row and column interchanged with row and column j, and D(j) is the scaling factor applied to row and column j, then SCALE(J) = P(J), for J = 1,...,ILO-1 = D(J), for J = ILO,...,IHI = P(J) for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
- **RCONE (output)**
  RCONE(j) is the reciprocal condition number of the j-th eigenvalue.
- **RCONV (output)**
  RCONV(j) is the reciprocal condition number of the j-th right eigenvector.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. If SENSE = 'N' or 'E', LDWORK >= max(1,2*N), and if JOBVL = 'V' or JOBVR = 'V', LDWORK >= 3*N. If SENSE = 'V' or 'B', LDWORK >= N*(N+6). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **IWORK2 (workspace)**
  dimension(2*N-2) If SENSE = 'N' or 'E', not referenced.
- **INFO (output)**

  = 0: successful exit

 < 0: if INFO = -i, the i-th argument had an illegal value.

 > 0: if INFO = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors or condition numbers
have been computed; elements 1:ILO-1 and i+1:N of WR
and WI contain eigenvalues which have converged.

NAME

dgegs - routine is deprecated and has been replaced by routine SGGES

SYNOPSIS

```
 SUBROUTINE DGEGS( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
*      BETA, VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 INTEGER N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)

 SUBROUTINE DGEGS_64( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHAR,
*      ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 INTEGER*8 N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGS( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 INTEGER :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR

 SUBROUTINE GEGS_64( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 INTEGER(8) :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void dgegs(char jobvsl, char jobvsr, int n, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *vsl, int ldvsl, double *vsr, int ldvsr, int *info);

void dgegs_64(char jobvsl, char jobvsr, long n, double *a, long lda, double *b, long ldb, double *alphar, double *alphai, double *beta, double *vsl, long ldvsl, double *vsr, long ldvsr, long *info);

PURPOSE

dgegs routine is deprecated and has been replaced by routine SGGES.

SGEGS computes for a pair of N-by-N real nonsymmetric matrices A, B: the generalized eigenvalues (alphar +/- alphai*i, beta), the real Schur form (A, B), and optionally left and/or right Schur vectors (VSL and VSR).

(If only the generalized eigenvalues are needed, use the driver SGEGV instead.)

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations'', by G. Golub & C. van Loan (Johns Hopkins U. Press)

The (generalized) Schur form of a pair of matrices is the result of multiplying both matrices on the left by one orthogonal matrix and both on the right by another orthogonal matrix, these two orthogonal matrices being chosen so as to bring the pair of matrices into (real) Schur form.

A pair of matrices A, B is in generalized real Schur form if B is upper triangular with non-negative diagonal and A is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of A will be ``standardized'' by making the corresponding elements of B have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in A and B will have a complex conjugate pair of generalized eigenvalues.

The left and right Schur vectors are the columns of VSL and VSR, respectively, where VSL and VSR are the orthogonal matrices which reduce A and B to Schur form:

Schur form of (A,B) = ((VSL)**T A (VSR), (VSL)**T B (VSR))

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of A. Note: to avoid overflow, the Frobenius norm of the matrix A should be less than the overflow threshold.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of B. Note: to avoid overflow, the Frobenius norm of the matrix B should be less than the overflow threshold.

- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).

- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i, j =1,...,N and BETA(j),j =1,...,N are the diagonals of the complex Schur form (A,B) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).

- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >=1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,4*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for SGEQRF, SORMQR, and SORGQR.) Then compute: NB -- MAX of the blocksizes for SGEQRF, SORMQR, and SORGQR The optimal LDWORK is 2*N + N*(NB+1).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from SGGBAL

 =N+2: error return from SGEQRF

 =N+3: error return from SORMQR

 =N+4: error return from SORGQR

 =N+5: error return from SGGHRD

 =N+6: error return from SHGEQZ (other than failed
iteration)
 =N+7: error return from SGGBAK (computing VSL)

 =N+8: error return from SGGBAK (computing VSR)

 =N+9: error return from SLASCL (various places)
```

NAME

dgegv - routine is deprecated and has been replaced by routine SGGEV

SYNOPSIS

```
 SUBROUTINE DGEGV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
*      BETA, VL, LDVL, VR, LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE DGEGV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR,
*      ALPHAI, BETA, VL, LDVL, VR, LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE GEGV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dgegv(char jobvl, char jobvr, int n, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *vl, int ldvl, double *vr, int ldvr, int *info);

void dgegv_64(char jobvl, char jobvr, long n, double *a, long lda, double *b, long ldb, double *alphar, double *alphai, double *beta, double *vl, long ldvl, double *vr, long ldvr, long *info);

PURPOSE

dgegv routine is deprecated and has been replaced by routine SGGEV.

SGEGV computes for a pair of n-by-n real nonsymmetric matrices A and B, the generalized eigenvalues (alphar +/- alphai*i, beta), and optionally, the left and/or right generalized eigenvectors (VL and VR).

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations'', by G. Golub & C. van Loan (Johns Hopkins U. Press)

A right generalized eigenvector corresponding to a generalized eigenvalue w for a pair of matrices (A,B) is a vector r such that (A - w B) r = 0 . A left generalized eigenvector is a vector l such that l**H * (A - w B) = 0, where l**H is the

conjugate-transpose of l.

Note: this routine performs ``full balancing'' on A and B -- see ``Further Details'', below.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```
- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```
- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of A on exit, see ``Further Details'', below.)
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of B on exit, see ``Further Details'', below.)
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose'', above.) Real eigenvectors take one column, complex take two columns, the first for the real part and the second for the imaginary part. Complex eigenvectors correspond to an eigenvalue with positive imaginary part. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL > = 1, and if JOBVL = 'V', LDVL > = N.

- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors. (See ``Purpose'', above.) Real eigenvectors take one column, complex take two columns, the first for the real part and the second for the imaginary part. Complex eigenvectors correspond to an eigenvalue with positive imaginary part. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR > = 1, and if JOBVR = 'V', LDVR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,8*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for SGEQRF, SORMQR, and SORGQR.) Then compute: NB -- MAX of the blocksizes for SGEQRF, SORMQR, and SORGQR; The optimal LDWORK is: 2*N + MAX(6*N, N*(NB+1)).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from SGGBAL

 =N+2: error return from SGEQRF

 =N+3: error return from SORMQR

 =N+4: error return from SORGQR

 =N+5: error return from SGGHRD

 =N+6: error return from SHGEQZ (other than failed
iteration)
 =N+7: error return from STGEVC

 =N+8: error return from SGGBAK (computing VL)

 =N+9: error return from SGGBAK (computing VR)

 =N+10: error return from SLASCL (various calls)
```

FURTHER DETAILS

Balancing

This driver calls SGGBAL to both permute and scale rows and columns of A and B. The permutations PL and PR are chosen so that PL*A*PR and PL*B*R will be upper triangular except for the diagonal blocks `A(i:j,i:j)` and B(i:j,i:j), with i and j as close together as possible. The diagonal scaling matrices DL and DR are chosen so that the pair DL*PL*A*PR*DR, DL*PL*B*PR*DR have elements close to one (except for the elements that start out zero.)

After the eigenvalues and eigenvectors of the balanced matrices have been computed, SGGBAK transforms the eigenvectors back to what they would have been (in perfect arithmetic) if they had not been balanced.

Contents of A and B on Exit

-------- -- - --- - -- ----

If any eigenvectors are computed (either JOBVL ='V' or JOBVR ='V' or both), then on exit the arrays A and B will contain the real Schur form[*] of the ``balanced'' versions of A and B. If no eigenvectors are computed, then only the diagonal blocks will be correct.

[*] See SHGEQZ, SGEGS, or read the book ``Matrix Computations'', by Golub & van Loan, pub. by Johns Hopkins U. Press.

NAME

dgehrd - reduce a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation

---

SYNOPSIS

```
  SUBROUTINE DGEHRD( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN, INFO)
  INTEGER N, ILO, IHI, LDA, LWORKIN, INFO
  DOUBLE PRECISION A(LDA,*), TAU(*), WORKIN(*)

  SUBROUTINE DGEHRD_64( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN,
 *      INFO)
  INTEGER*8 N, ILO, IHI, LDA, LWORKIN, INFO
  DOUBLE PRECISION A(LDA,*), TAU(*), WORKIN(*)
```

F95 INTERFACE

```
  SUBROUTINE GEHRD( [N], ILO, IHI, A, [LDA], TAU, [WORKIN], [LWORKIN],
 *       [INFO])
  INTEGER :: N, ILO, IHI, LDA, LWORKIN, INFO
  REAL(8), DIMENSION(:) :: TAU, WORKIN
  REAL(8), DIMENSION(:,:) :: A

  SUBROUTINE GEHRD_64( [N], ILO, IHI, A, [LDA], TAU, [WORKIN],
 *       [LWORKIN], [INFO])
  INTEGER(8) :: N, ILO, IHI, LDA, LWORKIN, INFO
  REAL(8), DIMENSION(:) :: TAU, WORKIN
  REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgehrd(int n, int ilo, int ihi, double *a, int lda, double *tau, int *info);

void dgehrd_64(long n, long ilo, long ihi, double *a, long lda, double *tau, long *info);

PURPOSE

dgehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation: Q' * A * Q = H .

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGEBAL; otherwise they should be set to 1 and N respectively. See Further Details.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details). Elements 1:ILO-1 and IHI:N-1 of TAU are set to zero.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The length of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of (ihi-ilo) elementary reflectors

```
Q  = H(ilo) H(ilo+1) . . . H(ihi-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0, `v(i+1)` = 1 and `v(ihi+1:n)` = 0; `v(i+2:ihi)` is stored on exit in A(i+2:ihi,i), and tau in TAU(i).

The contents of A are illustrated by the following example, with n = 7, ilo = 2 and ihi = 6:

on entry, on exit,

(a a a a a a a) (a a h h h h a) (a a a a a a) (a h h h h a) (a a a a a a) (h h h h h h) (a a a a a a) (v2 h h h h h) (a a a a a a) (v2 v3 h h h h) (a a a a a a) (v2 v3 v4 h h h) (a) (a)

where a denotes an element of the original matrix A, h denotes a modified element of the upper Hessenberg matrix H, and vi denotes an element of the vector defining H(i).

NAME

dgelqf - compute an LQ factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE DGELQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGELQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GELQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GELQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgelqf(int m, int n, double *a, int lda, double *tau, int *info);

void dgelqf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dgelqf computes an LQ factorization of a real M-by-N matrix A: A = L * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and below the diagonal of the array contain the m-by-min(m,n) lower trapezoidal matrix L (L is lower triangular if m <= n); the elements above the diagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i,i+1:n), and tau in TAU(i).

NAME

dgels - solve overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A

SYNOPSIS

```
 SUBROUTINE DGELS( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK, LDWORK,
*      INFO)
 CHARACTER * 1 TRANSA
 INTEGER M, N, NRHS, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE DGELS_64( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, NRHS, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELS( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GELS_64( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgels(char transa, int m, int n, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dgels_64(char transa, long m, long n, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dgels solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||.

2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B.

3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B.

4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

ARGUMENTS

- **TRANSA (input)**

  ```
  = 'N': the linear system involves A;

  = 'T': the linear system involves A**T.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > =0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M > = N, A is overwritten by details of its QR factorization as returned by SGEQRF; if M < N, A is overwritten by details of its LQ factorization as returned by SGELQF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the matrix B of right hand side vectors, stored columnwise; B is M-by-NRHS if TRANSA = 'N', or

N-by-NRHS if TRANSA = 'T'. On exit, B is overwritten by the solution vectors, stored columnwise: if TRANSA = 'N' and m >= n, rows 1 to n of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements N+1 to M in that column; if TRANSA = 'N' and m < n, rows 1 to N of B contain the minimum norm solution vectors; if TRANSA = 'T' and m >= n, rows 1 to M of B contain the minimum norm solution vectors; if TRANSA = 'T' and m < n, rows 1 to M of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements M+1 to N in that column.

- **LDB (input)**
  The leading dimension of the array B. LDB >= MAX(1,M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1, MN + max(MN, NRHS)). For optimal performance, LDWORK >= max(1, MN + max(MN, NRHS)*NB). where MN = `min(M,N)` and NB is the optimum block size.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dgelsd - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE DGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK,
*      LWORK, IWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), S(*), WORK(*)

 SUBROUTINE DGELSD_64( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK,
*      WORK, LWORK, IWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), S(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSD( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [IWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: S, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GELSD_64( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [IWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: S, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgelsd(int m, int n, int nrhs, double *a, int lda, double *b, int ldb, double *s, double rcond, int *rank, int *info);

void dgelsd_64(long m, long n, long nrhs, double *a, long lda, double *b, long ldb, double *s, double rcond, long *rank, long *info);

PURPOSE

dgelsd computes the minimum-norm solution to a real linear least squares problem: minimize 2-norm(| b - A*x |)

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The problem is solved in three steps:

(1) Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a ``bidiagonal least squares problem'' (BLS)

(2) Solve the BLS using a divide and conquer approach.

(3) Apply back all the Householder tranformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **M (input)**
  The number of rows of A. M >= 0.
- **N (input)**
  The number of columns of A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m >= n and RANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,max(M,N)).
- **S (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = S(1)/S(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `S(i)` <= RCOND*S(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **RANK (output)**
  The effective rank of A, i.e., the number of singular values which are greater than RCOND*S(1).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. The exact minimum amount of workspace needed depends on M, N and NRHS. As long as LWORK is at least 12*N + 2*N*SMLSIZ + 8*N*NLVL + N*NRHS * (SMLSIZ+1)**2, if M is greater than or equal to N

or 12*M + 2*M*SMLSIZ + 8*M*NLVL + M*NRHS + (SMLSIZ+1)**2, if M is less than N, the code will execute correctly. SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and NLVL = INT(LOG_2(MIN(M,N)/(SMLSIZ+1))) + 1 For good performance, LWORK should generally be larger.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  LIWORK > = 3 * MINMN * NLVL + 11 * MINMN, where MINMN = MIN(M,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Ren-Cang Li, Computer Science Division, University of California at Berkeley, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

dgelss - compute the minimum norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE DGELSS( M, N, NRHS, A, LDA, B, LDB, SING, RCOND, IRANK,
*      WORK, LDWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), SING(*), WORK(*)

 SUBROUTINE DGELSS_64( M, N, NRHS, A, LDA, B, LDB, SING, RCOND,
*      IRANK, WORK, LDWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), SING(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSS( [M], [N], [NRHS], A, [LDA], B, [LDB], SING, RCOND,
*       IRANK, [WORK], [LDWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SING, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GELSS_64( [M], [N], [NRHS], A, [LDA], B, [LDB], SING,
*       RCOND, IRANK, [WORK], [LDWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SING, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgelss(int m, int n, int nrhs, double *a, int lda, double *b, int ldb, double *sing, double rcond, int *irank, int *info);

void dgelss_64(long m, long n, long nrhs, double *a, long lda, double *b, long ldb, double *sing, double rcond, long *irank, long *info);

PURPOSE

dgelss computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm(| b - A*x |).

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the first `min(m,n)` rows of A are overwritten with its right singular vectors, stored rowwise.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,max(M,N)).
- **SING (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = SING(1)/SING(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `SING(i)` < = RCOND*SING(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **IRANK (output)**

The effective rank of A, i.e., the number of singular values which are greater than RCOND*SING(1).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.

- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1, and also: LDWORK >= 3*min(M,N) + max(2*min(M,N), max(M,N), NRHS) For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

NAME

dgelsx - routine is deprecated and has been replaced by routine SGELSY

SYNOPSIS

```
 SUBROUTINE DGELSX( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND, IRANK,
*      WORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER JPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE DGELSX_64( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND,
*      IRANK, WORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER*8 JPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSX( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER, DIMENSION(:) :: JPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GELSX_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER(8), DIMENSION(:) :: JPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgelsx(int m, int n, int nrhs, double *a, int lda, double *b, int ldb, int *jpivot, double rcond, int *irank, int *info);

void dgelsx_64(long m, long n, long nrhs, double *a, long lda, double *b, long ldb, long *jpivot, double rcond, long *irank, long *info);

PURPOSE

dgelsx routine is deprecated and has been replaced by routine SGELSY.

SGELSX computes the minimum-norm solution to a real linear least squares problem:

```
    minimize || A * X - B ||
```

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X. If m >= n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements N+1:M in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is an initial column, otherwise it is a free column. Before the QR factorization of A, all initial columns are permuted to the leading positions; only the remaining free columns are moved as a result of column pivoting during the factorization. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **IRANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  (max(min(M,N)+3*N, 2*min(M,N)+NRHS)),
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dgelsy - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE DGELSY( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER JPVT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE DGELSY_64( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 JPVT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSY( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT, RCOND,
*       RANK, [WORK], [LWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GELSY_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT,
*       RCOND, RANK, [WORK], [LWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: JPVT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgelsy(int m, int n, int nrhs, double *a, int lda, double *b, int ldb, int *jpvt, double rcond, int *rank, int *info);

void dgelsy_64(long m, long n, long nrhs, double *a, long lda, double *b, long ldb, long *jpvt, double rcond, long *rank, long *info);

PURPOSE

dgelsy computes the minimum-norm solution to a real linear least squares problem: minimize || A * X - B ||

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

This routine is basically identical to the original xGELSX except three differences:

```
  o The call to the subroutine xGEQPF has been substituted by the
    the call to the subroutine xGEQP3. This subroutine is a Blas-3
    version of the QR factorization with column pivoting.
  o Matrix B (the right hand side) is updated with Blas-3.
  o The permutation of matrix B (the right hand side) is faster and
    more simple.
```

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPVT (input/output)**
  On entry, if `JPVT(i)` .ne. 0, the i-th column of A is permuted to the front of AP, otherwise column i is a free column. On exit, if `JPVT(i)` = k, then the i-th column of AP was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **RANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. The unblocked strategy requires that: LWORK >= MAX(MN+3*N+1, 2*MN+NRHS), where MN = min(M, N). The block algorithm requires that: LWORK >= MAX(MN+2*N+NB*(N+1), 2*MN+NB*NRHS), where NB is an upper bound on the blocksize returned by ILAENV for the routines SGEQP3, STZRZF, STZRQF, SORMQR, and SORMRZ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0: successful exit

< 0: If INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
E. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain

---

NAME

dgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

SYNOPSIS

```
 SUBROUTINE DGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 INTEGER M, N, K, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE DGEMM_64( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 INTEGER*8 M, N, K, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE GEMM( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A, [LDA],
*       B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 INTEGER :: M, N, K, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C

 SUBROUTINE GEMM_64( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A,
*       [LDA], B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 INTEGER(8) :: M, N, K, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void dgemm(char transa, char transb, int m, int n, int k, double alpha, double *a, int lda, double *b, int ldb, double beta, double *c, int ldc);

void dgemm_64(char transa, char transb, long m, long n, long k, double alpha, double *a, long lda, double *b, long ldb, double beta, double *c, long ldc);

PURPOSE

dgemm performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C where op(X) is one of

```
op( X ) = X   or   op( X ) = X',
```

alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n', op(A) = A.

  TRANSA = 'T' or 't', op(A) = A'.

  TRANSA = 'C' or 'c', op(A) = A'.

  Unchanged on exit.
- **TRANSB (input)**
  On entry, TRANSB specifies the form of op(B) to be used in the matrix multiplication as follows:

  TRANSB = 'N' or 'n', op(B) = B.

  TRANSB = 'T' or 't', op(B) = B'.

  TRANSB = 'C' or 'c', op(B) = B'.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix op(A) and of the matrix C. M must be at least zero. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K must be at least zero. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is m otherwise. Before entry with TRANSA = 'N' or 'n', the leading m by k part of the array A must contain the matrix A, otherwise the leading k by m part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA > = max(1, m), otherwise LDA > = max(1, k). Unchanged on exit.
- **B (input)**
  n when TRANSB = 'N' or 'n', and is k otherwise. Before entry with TRANSB = 'N' or 'n', the leading k by n part of the array B must contain the matrix B, otherwise the leading n by k part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSB = 'N' or 'n' then LDB > = max(1, k), otherwise LDB > = max(1, n). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n matrix (alpha*op(A)*op(B) + beta*C).
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

dgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE DGEMV( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 INTEGER M, N, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)

 SUBROUTINE DGEMV_64( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE GEMV( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE GEMV_64( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgemv(char transa, int m, int n, double alpha, double *a, int lda, double *x, int incx, double beta, double *y, int incy);

void dgemv_64(char transa, long m, long n, double alpha, double *a, long lda, double *x, long incx, double beta, double *y, long incy);

PURPOSE

dgemv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*A'*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry with BETA non-zero, the incremented array Y must contain the vector y. On exit, Y is overwritten by the

updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dgeqlf - compute a QL factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE DGEQLF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGEQLF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQLF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GEQLF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgeqlf(int m, int n, double *a, int lda, double *tau, int *info);

void dgeqlf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dgeqlf computes a QL factorization of a real M-by-N matrix A: A = Q * L.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m >= n, the lower triangle of the subarray `A(m-n+1:m,1:n)` contains the N-by-N lower triangular matrix L; if m <= n, the elements on and below the (n-m)-th superdiagonal contain the M-by-N lower trapezoidal matrix L; the remaining elements, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(m-k+i+1:m)` = 0 and `v(m-k+i)` = 1; `v(1:m-k+i-1)` is stored on exit in A(1:m-k+i-1,n-k+i), and tau in TAU(i).

NAME

dgeqp3 - compute a QR factorization with column pivoting of a matrix A

SYNOPSIS

```
SUBROUTINE DGEQP3( M, N, A, LDA, JPVT, TAU, WORK, LWORK, INFO)
INTEGER M, N, LDA, LWORK, INFO
INTEGER JPVT(*)
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGEQP3_64( M, N, A, LDA, JPVT, TAU, WORK, LWORK, INFO)
INTEGER*8 M, N, LDA, LWORK, INFO
INTEGER*8 JPVT(*)
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQP3( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [INFO])
INTEGER :: M, N, LDA, LWORK, INFO
INTEGER, DIMENSION(:) :: JPVT
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GEQP3_64( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LWORK, INFO
INTEGER(8), DIMENSION(:) :: JPVT
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgeqp3(int m, int n, double *a, int lda, int *jpvt, double *tau, int *info);

void dgeqp3_64(long m, long n, double *a, long lda, long *jpvt, double *tau, long *info);

PURPOSE

dgeqp3 computes a QR factorization with column pivoting of a matrix A: A*P = Q*R using Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper trapezoidal matrix R; the elements below the diagonal, together with the array TAU, represent the orthogonal matrix Q as a product of `min(M,N)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **JPVT (input/output)**
  On entry, if JPVT(J).ne.0, the J-th column of A is permuted to the front of A*P (a leading column); if `JPVT(J)` =0, the J-th column of A is a free column. On exit, if `JPVT(J)` =K, then the J-th column of A*P was the the K-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 3*N+1. For optimal performance LWORK > = 2*N+(N+1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0: successful exit.

  < 0: if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real/complex scalar, and v is a real/complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

Based on contributions by

```
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
X. Sun, Computer Science Dept., Duke University, USA
```

NAME

dgeqpf - routine is deprecated and has been replaced by routine SGEQP3

SYNOPSIS

```
SUBROUTINE DGEQPF( M, N, A, LDA, JPIVOT, TAU, WORK, INFO)
INTEGER M, N, LDA, INFO
INTEGER JPIVOT(*)
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGEQPF_64( M, N, A, LDA, JPIVOT, TAU, WORK, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 JPIVOT(*)
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQPF( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: JPIVOT
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GEQPF_64( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: JPIVOT
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgeqpf(int m, int n, double *a, int lda, int *jpivot, double *tau, int *info);

void dgeqpf_64(long m, long n, double *a, long lda, long *jpivot, double *tau, long *info);

PURPOSE

dgeqpf routine is deprecated and has been replaced by routine SGEQP3.

SGEQPF computes a QR factorization with column pivoting of a real M-by-N matrix A: A*P = Q*R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper triangular matrix R; the elements below the diagonal, together with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is permuted to the front of A*P (a leading column); if `JPIVOT(i)` = 0, the i-th column of A is a free column. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n)
```

Each `H(i)` has the form

```
H  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i).

The matrix P is represented in jpvt as follows: If

```
jpvt(j)  = i
```

then the jth column of P is the ith canonical unit vector.

NAME

dgeqrf - compute a QR factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE DGEQRF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGEQRF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQRF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GEQRF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgeqrf(int m, int n, double *a, int lda, double *tau, int *info);

void dgeqrf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dgeqrf computes a QR factorization of a real M-by-N matrix A: A = Q * R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and above the diagonal of the array contain the min(M,N)-by-N upper trapezoidal matrix R (R is upper triangular if m >= n); the elements below the diagonal, with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

NAME

dger - perform the rank 1 operation A := alpha*x*y' + A

SYNOPSIS

```
SUBROUTINE DGER( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
INTEGER M, N, INCX, INCY, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(LDA,*)

SUBROUTINE DGER_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
INTEGER*8 M, N, INCX, INCY, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GER( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
INTEGER :: M, N, INCX, INCY, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GER_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
INTEGER(8) :: M, N, INCX, INCY, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dger(int m, int n, double alpha, double *x, int incx, double *y, int incy, double *a, int lda);

void dger_64(long m, long n, double alpha, double *x, long incx, double *y, long incy, double *a, long lda);

PURPOSE

dger performs the rank 1 operation A := alpha*x*y' + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

dgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DGERFS( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*       LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DGERFS_64( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*       LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GERFS( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GERFS_64( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dgerfs(char transa, int n, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dgerfs_64(char transa, long n, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dgerfs improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original N-by-N matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGETRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dgerqf - compute an RQ factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE DGERQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DGERQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GERQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GERQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgerqf(int m, int n, double *a, int lda, double *tau, int *info);

void dgerqf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dgerqf computes an RQ factorization of a real M-by-N matrix A: A = R * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m <= n, the upper triangle of the subarray `A(1:m,n-m+1:n)` contains the M-by-M upper triangular matrix R; if m >= n, the elements on and above the (m-n)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and tau in TAU(i).

NAME

dgesdd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and right singular vectors

---

SYNOPSIS

```
 SUBROUTINE DGESDD( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION A(LDA,*), S(*), U(LDU,*), VT(LDVT,*), WORK(*)

 SUBROUTINE DGESDD_64( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER*8 M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION A(LDA,*), S(*), U(LDU,*), VT(LDVT,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESDD( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT, [LDVT],
*       [WORK], [LWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: S, WORK
 REAL(8), DIMENSION(:,:) :: A, U, VT

 SUBROUTINE GESDD_64( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT,
*       [LDVT], [WORK], [LWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: S, WORK
```

```
REAL(8), DIMENSION(:,:) :: A, U, VT
```

C INTERFACE

#include <sunperf.h>

void dgesdd(char jobz, int m, int n, double *a, int lda, double *s, double *u, int ldu, double *vt, int ldvt, int *info);

void dgesdd_64(char jobz, long m, long n, double *a, long lda, double *s, double *u, long ldu, double *vt, long ldvt, long *info);

PURPOSE

dgesdd computes the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and right singular vectors. If singular vectors are desired, it uses a divide-and-conquer algorithm.

The SVD is written

```
 = U * SIGMA * transpose(V)
```

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M orthogonal matrix, and V is an N-by-N orthogonal matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns VT = V**T, not V.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U and all N rows of V**T are
returned in the arrays U and VT;
 = 'S':  the first min(M,N) columns of U and the first
min(M,N) rows of V**T are returned in the arrays U
and VT;
 = 'O':  If M  > = N, the first N columns of U are overwritten
on the array A and all rows of V**T are returned in
the array VT;
otherwise, all columns of U are returned in the
array U and the first M rows of V**T are overwritten
in the array VT;
 = 'N':  no columns of U or rows of V**T are computed.
```

- **M (input)**

The number of rows of the input matrix A. M >= 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBZ = 'O', A is overwritten with the first N columns of U (the left singular vectors, stored columnwise) if M >= N; A is overwritten with the first M rows of V**T (the right singular vectors, stored rowwise) otherwise. if JOBZ .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **S (output)**
  The singular values of A, sorted so that `S(i)` >= S(i+1).
- **U (output)**
  UCOL = M if JOBZ = 'A' or JOBZ = 'O' and M < N; UCOL = `min(M,N)` if JOBZ = 'S'. If JOBZ = 'A' or JOBZ = 'O' and M < N, U contains the M-by-M orthogonal matrix U; if JOBZ = 'S', U contains the first `min(M,N)` columns of U (the left singular vectors, stored columnwise); if JOBZ = 'O' and M >= N, or JOBZ = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBZ = 'S' or 'A' or JOBZ = 'O' and M < N, LDU >= M.
- **VT (output)**
  If JOBZ = 'A' or JOBZ = 'O' and M >= N, VT contains the N-by-N orthogonal matrix V**T; if JOBZ = 'S', VT contains the first `min(M,N)` rows of V**T (the right singular vectors, stored rowwise); if JOBZ = 'O' and M < N, or JOBZ = 'N', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBZ = 'A' or JOBZ = 'O' and M >= N, LDVT >= N; if JOBZ = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK;
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. If JOBZ = 'N', LWORK >= 3*min(M,N) + max(max(M,N),6*min(M,N)). If JOBZ = 'O', LWORK >= 3*min(M,N)*min(M,N) + max(max(M,N),5*min(M,N)* min(M,N)+4*min(M,N)). If JOBZ = 'S' or 'A' LWORK >= 3*min(M,N)*min(M,N) + max(max(M,N),4*min(M,N)* min(M,N)+4*min(M,N)). For good performance, LWORK should generally be larger. If LWORK < 0 but other input arguments are legal, `WORK(1)` returns optimal LWORK.
- **IWORK (workspace)**
  `dimension(8*MIN(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.

> 0:  SBDSDC did not converge, updating process failed.
```

FURTHER DETAILS

Based on contributions by

Ming Gu and Huan Ren, Computer Science Division, University of California at Berkeley, USA

NAME

dgesv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DGESV_64( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GESV( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE GESV_64( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgesv(int n, int nrhs, double *a, int lda, int *ipivot, double *b, int ldb, int *info);

void dgesv_64(long n, long nrhs, double *a, long lda, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dgesv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as

```
A = P * L * U,
```

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N coefficient matrix A. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution could not be computed.
```

NAME

dgesvd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors

---

SYNOPSIS

```
 SUBROUTINE DGESVD( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBU, JOBVT
 INTEGER M, N, LDA, LDU, LDVT, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), SING(*), U(LDU,*), VT(LDVT,*), WORK(*)

 SUBROUTINE DGESVD_64( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBU, JOBVT
 INTEGER*8 M, N, LDA, LDU, LDVT, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), SING(*), U(LDU,*), VT(LDVT,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVD( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 INTEGER :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL(8), DIMENSION(:) :: SING, WORK
 REAL(8), DIMENSION(:,:) :: A, U, VT

 SUBROUTINE GESVD_64( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL(8), DIMENSION(:) :: SING, WORK
 REAL(8), DIMENSION(:,:) :: A, U, VT
```

C INTERFACE

#include <sunperf.h>

void dgesvd(char jobu, char jobvt, int m, int n, double *a, int lda, double *sing, double *u, int ldu, double *vt, int ldvt, int *info);

void dgesvd_64(char jobu, char jobvt, long m, long n, double *a, long lda, double *sing, double *u, long ldu, double *vt, long ldvt, long *info);

PURPOSE

dgesvd computes the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors. The SVD is written = U * SIGMA * `transpose(V)`

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M orthogonal matrix, and V is an N-by-N orthogonal matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns V**T, not V.

ARGUMENTS

- **JOBU (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U are returned in array U:

 = 'S':  the first min(m,n) columns of U (the left singular
vectors) are returned in the array U;
 = 'O':  the first min(m,n) columns of U (the left singular
vectors) are overwritten on the array A;
 = 'N':  no columns of U (no left singular vectors) are
computed.
```

- **JOBVT (input)**
  Specifies options for computing all or part of the matrix V**T:

```
 = 'A':  all N rows of V**T are returned in the array VT;

 = 'S':  the first min(m,n) rows of V**T (the right singular
vectors) are returned in the array VT;
 = 'O':  the first min(m,n) rows of V**T (the right singular
vectors) are overwritten on the array A;
 = 'N':  no rows of V**T (no right singular vectors) are
computed.
```

  JOBVT and JOBU cannot both be 'O'.

- **M (input)**

The number of rows of the input matrix A. M >= 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBU = 'O', A is overwritten with the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBVT = 'O', A is overwritten with the first `min(m,n)` rows of V**T (the right singular vectors, stored rowwise); if JOBU .ne. 'O' and JOBVT .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **SING (output)**
  The singular values of A, sorted so that `SING(i)` >= SING(i+1).
- **U (output)**
  (LDU,M) if JOBU = 'A' or (LDU,min(M,N)) if JOBU = 'S'. If JOBU = 'A', U contains the M-by-M orthogonal matrix U; if JOBU = 'S', U contains the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBU = 'N' or 'O', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBU = 'S' or 'A', LDU >= M.
- **VT (output)**
  If JOBVT = 'A', VT contains the N-by-N orthogonal matrix V**T; if JOBVT = 'S', VT contains the first `min(m,n)` rows of V**T (the right singular vectors, stored rowwise); if JOBVT = 'N' or 'O', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBVT = 'A', LDVT >= N; if JOBVT = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK; if INFO > 0, `WORK(2:MIN(M,N))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in SING (not necessarily sorted). B satisfies A = U * B * VT, so it has the same singular values as A, and singular vectors related by U and VT.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1. LDWORK >= MAX(3*MIN(M,N)+MAX(M,N),5*MIN(M,N)). For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if SBDSQR did not converge, INFO specifies how many
superdiagonals of an intermediate bidiagonal form B
did not converge to zero. See the description of WORK
above for details.
```

NAME

dgesvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DGESVX( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR, WORK,
*      WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DGESVX_64( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF,
*      IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVX( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GESVX_64( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND,
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dgesvx(char fact, char transa, int n, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, char equed, double *rowsc, double *colsc, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dgesvx_64(char fact, char transa, long n, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, char equed, double *rowsc, double *colsc, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dgesvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = P * L * U,

where P is a permutation matrix, L is a unit lower triangular
matrix, and U is upper triangular.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

```
 = 'E':  The matrix A will be equilibrated if necessary, then
copied to AF and factored.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
 = 'N':  A * X  = B     (No transpose)

 = 'T':  A**T * X  = B  (Transpose)

 = 'COLSC':  A**H * X  = B  (Transpose)
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

EQUED = 'COLSC': A : = A * `diag(COLSC)`

EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the factors L and U from the factorization A = P*L*U as computed by SGETRF. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).

- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = P*L*U as computed by SGETRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the equilibrated matrix A.

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.

- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).

- **WORK (workspace)**
  `dimension(4*N)` On exit, `WORK(1)` contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If `WORK(1)` is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the

solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then WORK(1) contains the reciprocal pivot growth factor for the leading INFO columns of A.

- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization has
been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

dgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE DGETF2( M, N, A, LDA, IPIV, INFO)
INTEGER M, N, LDA, INFO
INTEGER IPIV(*)
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DGETF2_64( M, N, A, LDA, IPIV, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GETF2( [M], [N], A, [LDA], IPIV, [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GETF2_64( [M], [N], A, [LDA], IPIV, [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgetf2(int m, int n, double *a, int lda, int *ipiv, int *info);

void dgetf2_64(long m, long n, double *a, long lda, long *ipiv, long *info);

PURPOSE

dgetf2 computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 2 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the m by n matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **IPIV (output)**
  The pivot indices; for 1 < = i < = min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

dgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE DGETRF( M, N, A, LDA, IPIVOT, INFO)
INTEGER M, N, LDA, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DGETRF_64( M, N, A, LDA, IPIVOT, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GETRF( [M], [N], A, [LDA], IPIVOT, [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GETRF_64( [M], [N], A, [LDA], IPIVOT, [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgetrf(int m, int n, double *a, int lda, int *ipivot, int *info);

void dgetrf_64(long m, long n, double *a, long lda, long *ipivot, long *info);

PURPOSE

dgetrf computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 3 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

dgetri - compute the inverse of a matrix using the LU factorization computed by SGETRF

SYNOPSIS

```
SUBROUTINE DGETRI( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)

SUBROUTINE DGETRI_64( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GETRI( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE GETRI_64( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dgetri(int n, double *a, int lda, int *ipivot, int *info);

void dgetri_64(long n, double *a, long lda, long *ipivot, long *info);

PURPOSE

dgetri computes the inverse of a matrix using the LU factorization computed by SGETRF.

This method inverts U and then computes `inv(A)` by solving the system inv(A)*L = `inv(U)` for inv(A).

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the factors L and U from the factorization A = P*L*U as computed by SGETRF. On exit, if INFO = 0, the inverse of the original matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **WORK (workspace)**
  On exit, if INFO =0, then `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimal performance LDWORK > = N*NB, where NB is the optimal blocksize returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero; the matrix is
singular and its inverse could not be computed.
```

NAME

dgetrs - solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF

SYNOPSIS

```
SUBROUTINE DGETRS( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DGETRS_64( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GETRS( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: TRANSA
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE GETRS_64( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
CHARACTER(LEN=1) :: TRANSA
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgetrs(char transa, int n, int nrhs, double *a, int lda, int *ipivot, double *b, int ldb, int *info);

void dgetrs_64(char transa, long n, long nrhs, double *a, long lda, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dgetrs solves a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dggbak - form the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL

SYNOPSIS

```
 SUBROUTINE DGGBAK( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*), V(LDV,*)

 SUBROUTINE DGGBAK_64( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*), V(LDV,*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAK( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M], V,
*       [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE
 REAL(8), DIMENSION(:,:) :: V

 SUBROUTINE GGBAK_64( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M],
*       V, [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE
 REAL(8), DIMENSION(:,:) :: V
```

C INTERFACE

#include <sunperf.h>

void dggbak(char job, char side, int n, int ilo, int ihi, double *lscale, double *rscale, int m, double *v, int ldv, int *info);

void dggbak_64(char job, char side, long n, long ilo, long ihi, double *lscale, double *rscale, long m, double *v, long ldv, long *info);

PURPOSE

dggbak forms the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required:

```
= 'N':  do nothing, return immediately;

= 'P':  do backward transformation for permutation only;

= 'S':  do backward transformation for scaling only;

= 'B':  do backward transformations for both permutation and
scaling.
JOB must be the same as the argument JOB supplied to SGGBAL.
```

- **SIDE (input)**

```
= 'R':  V contains right eigenvectors;

= 'L':  V contains left eigenvectors.
```

- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by SGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description for ILO.
- **LSCALE (input)**
  Details of the permutations and/or scaling factors applied to the left side of A and B, as returned by SGGBAL.
- **RSCALE (input)**
  Details of the permutations and/or scaling factors applied to the right side of A and B, as returned by SGGBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**

On entry, the matrix of right or left eigenvectors to be transformed, as returned by STGEVC. On exit, V is overwritten by the transformed eigenvectors.

- **LDV (input)**
  The leading dimension of the matrix V. LDV >= max(1,N).
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. Ward, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

dggbal - balance a pair of general real matrices (A,B)

SYNOPSIS

```
 SUBROUTINE DGGBAL( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE, RSCALE,
*      WORK, INFO)
 CHARACTER * 1 JOB
 INTEGER N, LDA, LDB, ILO, IHI, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), LSCALE(*), RSCALE(*), WORK(*)

 SUBROUTINE DGGBAL_64( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE,
*      RSCALE, WORK, INFO)
 CHARACTER * 1 JOB
 INTEGER*8 N, LDA, LDB, ILO, IHI, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), LSCALE(*), RSCALE(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAL( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 INTEGER :: N, LDA, LDB, ILO, IHI, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GGBAL_64( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 INTEGER(8) :: N, LDA, LDB, ILO, IHI, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dggbal(char job, int n, double *a, int lda, double *b, int ldb, int *ilo, int *ihi, double *lscale, double *rscale, int *info);

void dggbal_64(char job, long n, double *a, long lda, double *b, long ldb, long *ilo, long *ihi, double *lscale, double *rscale, long *info);

PURPOSE

dggbal balances a pair of general real matrices (A,B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to ILO$-$1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem A*x = lambda*B*x.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A and B:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, LSCALE(I)  = 1.0
and RSCALE(I)  = 1.0 for i  = 1,...,N.
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the input matrix B. On exit, B is overwritten by the balanced matrix. If JOB = 'N', B is not referenced.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description for ILO.
- **LSCALE (input)**

Details of the permutations and scaling factors applied to the left side of A and B. If `P(j)` is the index of the row interchanged with row j, and `D(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.

- **RSCALE (input)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `P(j)` is the index of the column interchanged with column j, and `D(j)` is the scaling factor applied to column j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. WARD, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

dgges - compute for a pair of N-by-N real nonsymmetric matrices (A,B),

---

SYNOPSIS

```
 SUBROUTINE DGGES( JOBVSL, JOBVSR, SORT, DELCTG, N, A, LDA, B, LDB,
*      SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK,
*      BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL DELCTG
 LOGICAL BWORK(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)

 SUBROUTINE DGGES_64( JOBVSL, JOBVSR, SORT, DELCTG, N, A, LDA, B,
*      LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK,
*      LWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL*8 DELCTG
 LOGICAL*8 BWORK(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGES( JOBVSL, JOBVSR, SORT, DELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       [WORK], [LWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL :: DELCTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR

 SUBROUTINE GGES_64( JOBVSL, JOBVSR, SORT, DELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       [WORK], [LWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL(8) :: DELCTG
 LOGICAL(8), DIMENSION(:) :: BWORK
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void dgges(char jobvsl, char jobvsr, char sort, logical(*delctg)(double,double,double), int n, double *a, int lda, double *b, int ldb, int *sdim, double *alphar, double *alphai, double *beta, double *vsl, int ldvsl, double *vsr, int ldvsr, int *info);

void dgges_64(char jobvsl, char jobvsr, char sort, logical(*delctg)(double,double,double), long n, double *a, long lda, double *b, long ldb, long *sdim, double *alphar, double *alphai, double *beta, double *vsl, long ldvsl, double *vsr, long ldvsr, long *info);

PURPOSE

dgges computes for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized real Schur form (S,T), optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization

```
(A,B) = ( (VSL)*S*(VSR)**T, (VSL)*T*(VSR)**T )
```

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T.The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver SGGEV instead, which is faster.)

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or both being zero.

A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be ``standardized'' by making the corresponding elements of T have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues.

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

  ```
  = 'S':  Eigenvalues are ordered (see DELCTG);
  ```

- **DELCTG (input)**
  DELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', DELCTG is not referenced. If SORT = 'S', DELCTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue (ALPHAR(j)+ALPHAI(j))/BETA(j) is selected if `DELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` is true; i.e. if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

  Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy DELCTG(ALPHAR(j),ALPHAI(j), `BETA(j))` = .TRUE. after ordering. INFO is to be set to N+2 in this case.

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**

On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.

- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which DELCTG is true. (Complex conjugate pairs for which DELCTG is true for either eigenvalue count as 2.)
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i, and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >=1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 8*N+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy DELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering failed in STGSEN.
```

NAME

dggesx - compute for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and,

---

SYNOPSIS

```
  SUBROUTINE DGGESX( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, N, A, LDA,
 *      B, LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR,
 *      RCONDE, RCONDV, WORK, LWORK, IWORK, LIWORK, BWORK, INFO)
  CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
  INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
  INTEGER IWORK(*)
  LOGICAL DELCTG
  LOGICAL BWORK(*)
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), RCONDE(*), RCONDV(*),
WORK(*)

  SUBROUTINE DGGESX_64( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, N, A,
 *      LDA, B, LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR,
 *      RCONDE, RCONDV, WORK, LWORK, IWORK, LIWORK, BWORK, INFO)
  CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
  INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
  INTEGER*8 IWORK(*)
  LOGICAL*8 DELCTG
  LOGICAL*8 BWORK(*)
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), RCONDE(*), RCONDV(*),
WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GGESX( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, [N], A, [LDA],
 *       B, [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
 *       RCONDE, RCONDV, [WORK], [LWORK], [IWORK], [LIWORK], [BWORK],
 *       [INFO])
  CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
  INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL :: DELCTG
  LOGICAL, DIMENSION(:) :: BWORK
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, RCONDE, RCONDV, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR

  SUBROUTINE GGESX_64( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, [N], A,
 *       [LDA], B, [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR,
 *       [LDVSR], RCONDE, RCONDV, [WORK], [LWORK], [IWORK], [LIWORK],
 *       [BWORK], [INFO])
  CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
  INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8) :: DELCTG
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, RCONDE, RCONDV, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void dggesx(char jobvsl, char jobvsr, char sort, logical(*delctg)(double,double,double), char sense, int n, double *a, int lda, double *b, int ldb, int *sdim, double *alphar, double *alphai, double *beta, double *vsl, int ldvsl, double *vsr, int ldvsr, double *rconde, double *rcondv, int *info);

void dggesx_64(char jobvsl, char jobvsr, char sort, logical(*delctg)(double,double,double), char sense, long n, double *a, long lda, double *b, long ldb, long *sdim, double

*alphar, double *alphai, double *beta, double *vsl, long ldvsl, double *vsr, long ldvsr, double *rconde, double *rcondv, long *info);

PURPOSE

dggesx computes for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and, optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization

A,B) = ((VSL) S (VSR)**T, (VSL) T (VSR)**T)

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (RCONDV). The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or for both being zero.

A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be ``standardized'' by making the corresponding elements of T have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues.

ARGUMENTS

- **JOBVSL (input)**

```
= 'N':  do not compute the left Schur vectors;

= 'V':  compute the left Schur vectors.
```

- **JOBVSR (input)**

```
= 'N':  do not compute the right Schur vectors;

= 'V':  compute the right Schur vectors.
```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S':  Eigenvalues are ordered (see DELCTG).
```

- **DELCTG (input)**
  DELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', DELCTG is not referenced. If SORT = 'S', DELCTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue (ALPHAR(j)+ALPHAI(j))/BETA(j) is selected if `DELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` is true; i.e. if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy `DELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+3.
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N' : None are computed;

```
= 'E' : Computed for average of selected eigenvalues only;

= 'V' : Computed for selected deflating subspaces only;

= 'B' : Computed for both.
If SENSE  = 'E', 'V', or 'B', SORT must equal 'S'.
```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **SDIM (output)**

If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which DELCTG is true. (Complex conjugate pairs for which DELCTG is true for either eigenvalue count as 2.)

- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHAR(j) + ALPHAI(j)*i and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If ALPHAI(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with ALPHAI(j+1) negative.

  Note: the quotients ALPHAR(j)/BETA(j) and ALPHAI(j)/BETA(j) may easily over- or underflow, and BETA(j) may even be zero. Thus, the user should avoid naively computing the ratio. However, ALPHAR and ALPHAI will be always less than and usually comparable with norm(A) in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >=1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', RCONDE(1) and RCONDE(2) contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', RCONDV(1) and RCONDV(2) contain the reciprocal condition numbers for the selected deflating subspaces. Not referenced if SENSE = 'N' or 'E'.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 8*(N+1)+16. If SENSE = 'E', 'V', or 'B', LWORK >= MAX(8*(N+1)+16, 2*SDIM*(N-SDIM)).
- **IWORK (workspace)**
  Not referenced if SENSE = 'N'.
- **LIWORK (input)**
  The dimension of the array WORK. LIWORK >= N+6.
- **BWORK (workspace)**
  dimension(N) Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy DELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering failed in STGSEN.
```

Further details = = = = = = = = = = = = = = =

An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

EPS * norm((A, B)) / RCONDE(1).

An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is

EPS * norm((A, B)) / RCONDV(2).

See LAPACK User's Guide, section 4.11 for more information.

NAME

dggev - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

SYNOPSIS

```
  SUBROUTINE DGGEV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
 *       BETA, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)
  CHARACTER * 1 JOBVL, JOBVR
  INTEGER N, LDA, LDB, LDVL, LDVR, LWORK, INFO
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

  SUBROUTINE DGGEV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR,
 *       ALPHAI, BETA, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)
  CHARACTER * 1 JOBVL, JOBVR
  INTEGER*8 N, LDA, LDB, LDVL, LDVR, LWORK, INFO
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GGEV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
 *       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [INFO])
  CHARACTER(LEN=1) :: JOBVL, JOBVR
  INTEGER :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VL, VR

  SUBROUTINE GGEV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
 *       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [INFO])
  CHARACTER(LEN=1) :: JOBVL, JOBVR
  INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dggev(char jobvl, char jobvr, int n, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *vl, int ldvl, double *vr, int ldvr, int *info);

void dggev_64(char jobvl, char jobvr, long n, double *a, long lda, double *b, long ldb, double *alphar, double *alphai, double *beta, double *vl, long ldvl, double *vr, long ldvr, long *info);

PURPOSE

dggev computes for a pair of N-by-N real nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j).
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B .
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N >= 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten.
- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)+i*VL(:,j+1)` and `u(j+1)` = VL(:,j)-i*VL(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part)+abs(imag. part) =1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**

If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)+i*VR(:,j+1)` and `v(j+1)` = VR(:,j)-i*VR(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part)+abs(imag. part) =1. Not referenced if JOBVR = 'N'.

- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,8*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: error return from STGEVC.
```

NAME

dggevx - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

SYNOPSIS

```
  SUBROUTINE DGGEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B, LDB,
 *      ALPHAR, ALPHAI, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE,
 *      RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, BWORK,
 *      INFO)
  CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
  INTEGER N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER IWORK(*)
  LOGICAL BWORK(*)
  DOUBLE PRECISION ABNRM, BBNRM
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), LSCALE(*), RSCALE(*),
RCONDE(*), RCONDV(*), WORK(*)

  SUBROUTINE DGGEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B,
 *      LDB, ALPHAR, ALPHAI, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE,
 *      RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, BWORK,
 *      INFO)
  CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
  INTEGER*8 N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER*8 IWORK(*)
  LOGICAL*8 BWORK(*)
  DOUBLE PRECISION ABNRM, BBNRM
  DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), LSCALE(*), RSCALE(*),
RCONDE(*), RCONDV(*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GGEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHAR, ALPHAI, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI,
 *       LSCALE, RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  INTEGER :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL, DIMENSION(:) :: BWORK
  REAL(8) :: ABNRM, BBNRM
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, LSCALE, RSCALE, RCONDE, RCONDV, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VL, VR

  SUBROUTINE GGEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHAR, ALPHAI, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI,
 *       LSCALE, RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL(8) :: ABNRM, BBNRM
  REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, LSCALE, RSCALE, RCONDE, RCONDV, WORK
  REAL(8), DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dggevx(char balanc, char jobvl, char jobvr, char sense, int n, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *vl, int ldvl, double *vr, int ldvr, int *ilo, int *ihi, double *lscale, double *rscale, double *abnrm, double *bbnrm, double *rconde, double *rcondv, int *info);

void dggevx_64(char balanc, char jobvl, char jobvr, char sense, long n, double *a, long lda, double *b, long ldb, double *alphar, double *alphai, double *beta, double *vl, long ldvl, double *vr, long ldvr, long *ilo, long *ihi, double *lscale, double *rscale, double *abnrm, double *bbnrm, double *rconde, double *rcondv, long *info);

PURPOSE

dggevx computes for a pair of N-by-N real nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, LSCALE, RSCALE, ABNRM, and BBNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right eigenvectors (RCONDV).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j) .
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B.
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **BALANC (input)**
  Specifies the balance option to be performed. = 'N': do not diagonally scale or permute;

  ```
  = 'P':  permute only;

  = 'S':  scale only;

  = 'B':  both permute and scale.
  ```
  Computed reciprocal condition numbers will be for the matrices after permuting and/or balancing. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```
- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': none are computed;

  ```
  = 'E': computed for eigenvalues only;

  = 'V': computed for eigenvectors only;

  = 'B': computed for eigenvalues and eigenvectors.
  ```
- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then A contains the first part of the real Schur

form of the ``balanced'' versions of the input A and B.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then B contains the second part of the real Schur form of the ``balanced'' versions of the input A and B.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio ALPHA/BETA. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)+i*VL(:,j+1)` and `u(j+1)` = VL(:,j)-i*VL(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL > = 1, and if JOBVL = 'V', LDVL > = N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)+i*VR(:,j+1)` and `v(j+1)` = VR(:,j)-i*VR(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR > = 1, and if JOBVR = 'V', LDVR > = N.
- **ILO (output)**
  ILO and IHI are integer values such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description of ILO.
- **LSCALE (output)**
  Details of the permutations and scaling factors applied to the left side of A and B. If `PL(j)` is the index of the row interchanged with row j, and `DL(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `PL(j)` for j = 1,...,ILO-1 = `DL(j)` for j = ILO,...,IHI = `PL(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (output)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `PR(j)` is the index of the column interchanged with column j, and `DR(j)` is the scaling factor applied to column j, then `RSCALE(j)` = `PR(j)` for j = 1,...,ILO-1 = `DR(j)` for j = ILO,...,IHI = `PR(j)` for j = IHI+1,...,N The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix A.
- **BBNRM (output)**
  The one-norm of the balanced matrix B.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of RCONDE are set to the same value. Thus RCONDE(j), RCONDV(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If SENSE = 'V', RCONDE is not referenced.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of RCONDV are set to the same value. If the eigenvalues cannot be reordered to compute RCONDV(j), `RCONDV(j)` is set to 0; this can only occur when the true value would be very small anyway. If SENSE = 'E', RCONDV is not referenced.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,6*N). If SENSE = 'E', LWORK > = 12*N. If SENSE = 'V' or 'B', LWORK > = 2*N*N+12*N+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(N+6)` If SENSE = 'E', IWORK is not referenced.
- **BWORK (workspace)**
  `dimension(N)` If SENSE = 'N', BWORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: error return from STGEVC.
```

FURTHER DETAILS

Balancing a matrix pair (A,B) includes, first, permuting rows and columns to isolate eigenvalues, second, applying diagonal similarity transformation to the rows and columns to make the rows and columns as close in norm as possible. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.11.1.2 of LAPACK Users' Guide.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(ABNRM, BBNRM) / RCONDE(I)

An approximate error bound for the angle between the i-th computed eigenvector VL(i) or VR(i) is given by

PS * norm(ABNRM, BBNRM) / DIF(i).

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see section 4.11 of LAPACK User's Guide.

NAME

dggglm - solve a general Gauss-Markov linear model (GLM) problem

---

SYNOPSIS

```
 SUBROUTINE DGGGLM( N, M, P, A, LDA, B, LDB, D, X, Y, WORK, LDWORK,
*      INFO)
 INTEGER N, M, P, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)

 SUBROUTINE DGGGLM_64( N, M, P, A, LDA, B, LDB, D, X, Y, WORK,
*      LDWORK, INFO)
 INTEGER*8 N, M, P, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGGLM( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y, [WORK],
*       [LDWORK], [INFO])
 INTEGER :: N, M, P, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: D, X, Y, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GGGLM_64( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y,
*       [WORK], [LDWORK], [INFO])
 INTEGER(8) :: N, M, P, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: D, X, Y, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dggglm(int n, int m, int p, double *a, int lda, double *b, int ldb, double *d, double *x, double *y, int *info);

void dggglm_64(long n, long m, long p, double *a, long lda, double *b, long ldb, double *d, double *x, double *y, long

*info);

PURPOSE

dggglm solves a general Gauss-Markov linear model (GLM) problem:

```
        minimize || y ||_2   subject to   d = A*x + B*y

            x
```

where A is an N-by-M matrix, B is an N-by-P matrix, and d is a given N-vector. It is assumed that M <= N <= M+P, and

```
           rank(A) = M    and    rank( A B ) = N.
```

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of A and B.

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

```
             minimize || inv(B)*(d-A*x) ||_2

                 x
```

where `inv(B)` denotes the inverse of B.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. 0 < = M < = N.
- **P (input)**
  The number of columns of the matrix B. P > = N-M.
- **A (input/output)**
  On entry, the N-by-M matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-P matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **D (input/output)**
  On entry, D is the left hand side of the GLM equation. On exit, D is destroyed.
- **X (output)**
  On exit, X and Y are the solutions of the GLM problem.
- **Y (output)**

See the description of X.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N+M+P). For optimum performance, LDWORK >= M+min(N,P)+max(N,P)*NB, where NB is an upper bound for the optimal blocksizes for SGEQRF, SGERQF, SORMQR and SORMRQ.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

dgghrd - reduce a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular

---

SYNOPSIS

```
 SUBROUTINE DGGHRD( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)

 SUBROUTINE DGGHRD_64( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
```

F95 INTERFACE

```
 SUBROUTINE GGHRD( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE GGHRD_64( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dgghrd(char compq, char compz, int n, int ilo, int ihi, double *a, int lda, double *b, int ldb, double *q, int ldq, double *z, int ldz, int *info);

void dgghrd_64(char compq, char compz, long n, long ilo, long ihi, double *a, long lda, double *b, long ldb, double *q, long ldq, double *z, long ldz, long *info);

PURPOSE

dgghrd reduces a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular: Q' * A * Z = H and Q' * B * Z = T, where H is upper Hessenberg, T is upper triangular, and Q and Z are orthogonal, and ' means transpose.

The orthogonal matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q1 and Z1, so that

1 * A * Z1' = (Q1*Q) * H * (Z1*Z)' 1 * B * Z1' = (Q1*Q) * T * (Z1*Z)'

ARGUMENTS

- **COMPQ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
orthogonal matrix Q is returned;
 = 'V': Q must contain an orthogonal matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **COMPZ (input)**

```
 = 'N': do not compute Z;

 = 'I': Z is initialized to the unit matrix, and the
orthogonal matrix Z is returned;
 = 'V': Z must contain an orthogonal matrix Z1 on entry,
and the product Z1*Z is returned.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGGBAL; otherwise they should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**

On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the rest is set to zero.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. On exit, the upper triangular matrix T = Q' B Z. The elements below the diagonal are set to zero.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **Q (input/output)**
  If COMPQ ='N': Q is not referenced.

  If COMPQ ='I': on entry, Q need not be set, and on exit it contains the orthogonal matrix Q, where Q' is the product of the Givens transformations which are applied to A and B on the left. If COMPQ ='V': on entry, Q must contain an orthogonal matrix Q1, and on exit this is overwritten by Q1*Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = N if COMPQ ='V' or 'I'; LDQ > = 1 otherwise.
- **Z (input/output)**
  If COMPZ ='N': Z is not referenced.

  If COMPZ ='I': on entry, Z need not be set, and on exit it contains the orthogonal matrix Z, which is the product of the Givens transformations which are applied to A and B on the right. If COMPZ ='V': on entry, Z must contain an orthogonal matrix Z1, and on exit this is overwritten by Z1*Z.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = N if COMPZ ='V' or 'I'; LDZ > = 1 otherwise.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

This routine reduces A to Hessenberg and B to triangular form by an unblocked reduction, as described in _Matrix_Computations_, by Golub and Van Loan (Johns Hopkins Press.)

NAME

dgglse - solve the linear equality-constrained least squares (LSE) problem

---

SYNOPSIS

```
 SUBROUTINE DGGLSE( M, N, P, A, LDA, B, LDB, C, D, X, WORK, LDWORK,
*      INFO)
 INTEGER M, N, P, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)

 SUBROUTINE DGGLSE_64( M, N, P, A, LDA, B, LDB, C, D, X, WORK,
*      LDWORK, INFO)
 INTEGER*8 M, N, P, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGLSE( [M], [N], [P], A, [LDA], B, [LDB], C, D, X, [WORK],
*       [LDWORK], [INFO])
 INTEGER :: M, N, P, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: C, D, X, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GGLSE_64( [M], [N], [P], A, [LDA], B, [LDB], C, D, X,
*       [WORK], [LDWORK], [INFO])
 INTEGER(8) :: M, N, P, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: C, D, X, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dgglse(int m, int n, int p, double *a, int lda, double *b, int ldb, double *c, double *d, double *x, int *info);

void dgglse_64(long m, long n, long p, double *a, long lda, double *b, long ldb, double *c, double *d, double *x, long *info);

PURPOSE

dgglse solves the linear equality-constrained least squares (LSE) problem:

```
minimize || c - A*x ||_2   subject to   B*x = d
```

where A is an M-by-N matrix, B is a P-by-N matrix, c is a given M-vector, and d is a given P-vector. It is assumed that

P <= N <= M+P, and

```
rank(B) = P and  rank( ( A ) ) = N.
                       ( ( B ) )
```

These conditions ensure that the LSE problem has a unique solution, which is obtained using a GRQ factorization of the matrices B and A.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **P (input)**
  The number of rows of the matrix B. 0 < = P < = N < = M+P.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **C (input/output)**
  On entry, C contains the right hand side vector for the least squares part of the LSE problem. On exit, the residual sum of squares for the solution is given by the sum of squares of elements N-P+1 to M of vector C.
- **D (input/output)**
  On entry, D contains the right hand side vector for the constrained equation. On exit, D is destroyed.
- **X (output)**
  On exit, X is the solution of the LSE problem.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,M+N+P). For optimum performance LDWORK > = P+min(M,N)+max(M,N)*NB, where NB is an upper bound for the optimal blocksizes for SGEQRF, SGERQF, SORMQR and SORMRQ.

If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

dggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

SYNOPSIS

```
 SUBROUTINE DGGQRF( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 INTEGER N, M, P, LDA, LDB, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)

 SUBROUTINE DGGQRF_64( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 INTEGER*8 N, M, P, LDA, LDB, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGQRF( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER :: N, M, P, LDA, LDB, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAUA, TAUB, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GGQRF_64( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER(8) :: N, M, P, LDA, LDB, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAUA, TAUB, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dggqrf(int n, int m, int p, double *a, int lda, double *taua, double *b, int ldb, double *taub, int *info);

void dggqrf_64(long n, long m, long p, double *a, long lda, double *taua, double *b, long ldb, double *taub, long *info);

PURPOSE

dggqrf computes a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B:

```
A = Q*R,        B = Q*T*Z,
```

where Q is an N-by-N orthogonal matrix, Z is a P-by-P orthogonal matrix, and R and T assume one of the forms:

if N >= M, R = (R11) M , or if N < M, R = (R11 R12) N, (0) N-M N M-N M

where R11 is upper triangular, and

if N <= P, T = (0 T12) N, or if N > P, T = (T11) N-P, P-N N (T21) P P

where T12 or T21 is upper triangular.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of inv(B)*A:

```
inv(B)*A = Z'*(inv(T)*R)
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the transpose of the matrix Z.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. M > = 0.
- **P (input)**
  The number of columns of the matrix B. P > = 0.
- **A (input)**
  On entry, the N-by-M matrix A. On exit, the elements on and above the diagonal of the array contain the min(N,M)-by-M upper trapezoidal matrix R (R is upper triangular if N > = M); the elements below the diagonal, with the array TAUA, represent the orthogonal matrix Q as a product of `min(N,M)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q (see Further Details).
- **B (input)**
  On entry, the N-by-P matrix B. On exit, if N < = P, the upper triangle of the subarray `B(1:N,P-N+1:P)` contains the N-by-N upper triangular matrix T; if N > P, the elements on and above the (N-P)-th subdiagonal contain the N-by-P upper trapezoidal matrix T; the remaining elements, with the array TAUB, represent the orthogonal matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the QR factorization of an N-by-M matrix, NB2 is the optimal blocksize for the RQ factorization of an N-by-P matrix, and NB3 is the optimal blocksize for a call of SORMQR.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(n,m).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i+1:n,i), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine SORGQR.

To use Q to update another matrix, use LAPACK subroutine SORMQR.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(n,p).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a real scalar, and v is a real vector with

`v(p-k+i+1:p)` = 0 and `v(p-k+i)` = 1; `v(1:p-k+i-1)` is stored on exit in B(n-k+i,1:p-k+i-1), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine SORGRQ.

To use Z to update another matrix, use LAPACK subroutine SORMRQ.

NAME

dggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

SYNOPSIS

```
 SUBROUTINE DGGRQF( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 INTEGER M, P, N, LDA, LDB, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)

 SUBROUTINE DGGRQF_64( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 INTEGER*8 M, P, N, LDA, LDB, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGRQF( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER :: M, P, N, LDA, LDB, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAUA, TAUB, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE GGRQF_64( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER(8) :: M, P, N, LDA, LDB, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAUA, TAUB, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dggrqf(int m, int p, int n, double *a, int lda, double *taua, double *b, int ldb, double *taub, int *info);

void dggrqf_64(long m, long p, long n, double *a, long lda, double *taua, double *b, long ldb, double *taub, long *info);

PURPOSE

dggrqf computes a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B:

```
A = R*Q,        B = Z*T*Q,
```

where Q is an N-by-N orthogonal matrix, Z is a P-by-P orthogonal matrix, and R and T assume one of the forms:

if M <= N, R = (0 R12) M, or if M > N, R = (R11) M-N, N-M M (R21) N N

where R12 or R21 is upper triangular, and

if P >= N, T = (T11) N , or if P < N, T = (T11 T12) P, (0) P-N P N-P N

where T11 is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of A*inv(B):

```
A*inv(B) = (R*inv(T))*Z'
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the transpose of the matrix Z.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **P (input)**
  The number of rows of the matrix B. P > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M < = N, the upper triangle of the subarray `A(1:M,N-M+1:N)` contains the M-by-M upper triangular matrix R; if M > N, the elements on and above the (M-N)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAUA, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q (see Further Details).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, the elements on and above the diagonal of the array contain the min(P,N)-by-N upper trapezoidal matrix T (T is upper triangular if P > = N); the elements below the diagonal, with the array TAUB, represent the orthogonal matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the RQ factorization of an M-by-N matrix, NB2 is the optimal blocksize for the QR factorization of a P-by-N matrix, and NB3 is the optimal blocksize for a call of SORMRQ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INF0 = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a real scalar, and v is a real vector with

`v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine SORGRQ.

To use Q to update another matrix, use LAPACK subroutine SORMRQ.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(p,n).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:p)` is stored on exit in B(i+1:p,i), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine SORGQR.

To use Z to update another matrix, use LAPACK subroutine SORMQR.

NAME

dggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B

SYNOPSIS

```
 SUBROUTINE DGGSVD( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B, LDB,
*      ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER IWORK3(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)

 SUBROUTINE DGGSVD_64( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B,
*      LDB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK3(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVD( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK3
 REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE GGSVD_64( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER(8) :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3
 REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void dggsvd(char jobu, char jobv, char jobq, int m, int n, int p, int *k, int *l, double *a, int lda, double *b, int ldb, double *alpha, double *beta, double *u, int ldu, double *v, int ldv, double *q, int ldq, int *iwork3, int *info);

void dggsvd_64(char jobu, char jobv, char jobq, long m, long n, long p, long *k, long *l, double *a, long lda, double *b, long ldb, double *alpha, double *beta, double *u, long ldu, double *v, long ldv, double *q, long ldq, long *iwork3, long *info);

PURPOSE

dggsvd computes the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B:

```
    U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R )
```

where U, V and Q are orthogonal matrices, and Z' is the transpose of Z. Let K+L = the effective numerical rank of the matrix (A',B')', then R is a K+L-by-K+L nonsingular upper triangular matrix, D1 and D2 are M-by-(K+L) and P-by-(K+L) ``diagonal'' matrices and of the following structures, respectively:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
            M-K-L ( 0  0 )

                  K  L
       D2 =   L ( 0  S )
            P-L ( 0  0 )

                N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 )
            L (  0    0   R22 )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),

  S = diag( BETA(K+1),  ... , BETA(K+L) ),

  C**2 + S**2 = I.

  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                    K M-K K+L-M
```

```
       D1 =   K ( I  0    0   )

            M-K ( 0  C    0   )

                    K M-K K+L-M

       D2 =   M-K ( 0  S    0  )

            K+L-M ( 0  0    I  )

              P-L ( 0  0    0  )

                   N-K-L  K   M-K  K+L-M

  ( 0 R ) =     K ( 0    R11  R12  R13  )

              M-K ( 0     0   R22  R23  )

            K+L-M ( 0     0    0   R33  )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(M) ),

  S = diag( BETA(K+1),  ... , BETA(M) ),

  C**2 + S**2 = I.

  (R11 R12 R13 ) is stored in A(1:M, N-K-L+1:N), and R33 is stored
  ( 0  R22 R23 )

  in B(M-K+1:L,N+M-K-L+1:N) on exit.
```

The routine computes C, S, R, and optionally the orthogonal transformation matrices U, V and Q.

In particular, if B is an N-by-N nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A*inv(B):

```
                     A*inv(B) = U*(D1*inv(D2))*V'.
```

If (A',B')' has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem: A'*A x = lambda* B'*B x.

In some literature, the GSVD of A and B is presented in the form U'*A*X = (0 D1), V'*B*X = (0 D2)

where U and V are orthogonal and X is nonsingular, D1 and D2 are ``diagonal''. The former GSVD form can be converted to the latter form by taking the nonsingular matrix X as

```
                     X = Q*( I   0    )

                           ( 0 inv(R) ).
```

ARGUMENTS

- **JOBU (input)**

  = 'U': Orthogonal matrix U is computed;

  = 'N': U is not computed.
- **JOBV (input)**

  = 'V': Orthogonal matrix V is computed;

  = 'N': V is not computed.
- **JOBQ (input)**

  = 'Q': Orthogonal matrix Q is computed;

  = 'N': Q is not computed.
- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in the Purpose section. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular matrix R, or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix R if M-K-L < 0. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDA >= max(1,P).
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L >= 0, ALPHA(K+1:K+L) = C,

  BETA(K+1:K+L) = S, or if M-K-L < 0, ALPHA(K+1:M) =C, ALPHA(M+1:K+L) =0

  BETA(K+1:M) =S, BETA(M+1:K+L) =1 and ALPHA(K+L+1:N) = 0

  BETA(K+L+1:N) = 0
- **BETA (output)**
  See the description of ALPHA.
- **U (output)**
  If JOBU = 'U', U contains the M-by-M orthogonal matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,M) if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the P-by-P orthogonal matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV >= max(1,P) if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (output)**
  If JOBQ = 'Q', Q contains the N-by-N orthogonal matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **WORK (workspace)**
  dimension (max(3*N,M,P)+N)
- **IWORK3 (output)**
  `dimension(N)` On exit, IWORK3 stores the sorting information. More precisely, the following loop will sort ALPHA for I = K+1, `min(M,K+L)` swap `ALPHA(I)` and `ALPHA(IWORK3(I))` endfor such that `ALPHA(1)` >= `ALPHA(2)` >= ... >= ALPHA(N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = 1, the Jacobi-type procedure failed to
converge.  For further details, see subroutine STGSJA.
```

NAME

dggsvp - compute orthogonal matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

SYNOPSIS

```
 SUBROUTINE DGGSVP( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB, TOLA,
*      TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, TAU, WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)

 SUBROUTINE DGGSVP_64( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB,
*      TOLA, TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, TAU, WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVP( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B, [LDB],
*       TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK], [TAU],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE GGSVP_64( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B,
*       [LDB], TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK],
*       [TAU], [WORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
INTEGER(8) :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8) :: TOLA, TOLB
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void dggsvp(char jobu, char jobv, char jobq, int m, int p, int n, double *a, int lda, double *b, int ldb, double tola, double tolb, int *k, int *l, double *u, int ldu, double *v, int ldv, double *q, int ldq, int *info);

void dggsvp_64(char jobu, char jobv, char jobq, long m, long p, long n, double *a, long lda, double *b, long ldb, double tola, double tolb, long *k, long *l, double *u, long ldu, double *v, long ldv, double *q, long ldq, long *info);

PURPOSE

dggsvp computes orthogonal matrices U, V and Q such that L (0 0 A23)

```
          M-K-L ( 0     0    0  )

                N-K-L  K    L

        =     K ( 0    A12  A13 )  if M-K-L < 0;

            M-K ( 0     0   A23 )

                N-K-L  K    L

 V'*B*Q =   L ( 0     0   B13 )

          P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal. K+L = the effective numerical rank of the (M+P)-by-N matrix (A',B')'. Z' denotes the transpose of Z.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine SGGSVD.

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Orthogonal matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Orthogonal matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Orthogonal matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular (or trapezoidal) matrix described in the Purpose section.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix described in the Purpose section.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the thresholds to determine the effective numerical rank of matrix B and a subblock of A. Generally, they are set to TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS. The size of TOLA and TOLB may affect the size of backward errors of the decomposition.
- **TOLB (input)**
  See the description of TOLA.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **U (output)**
  If JOBU = 'U', U contains the orthogonal matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the orthogonal matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**

The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (output)**
  If JOBQ = 'Q', Q contains the orthogonal matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **IWORK (workspace)**
  `dimension(N)`
- **TAU (workspace)**
  `dimension(N)`
- **WORK (workspace)**
  `dimension(MAX(3*N,M,P))`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The subroutine uses LAPACK subroutine SGEQPF for the QR factorization with column pivoting to detect the effective numerical rank of the a matrix. It may be replaced by a better rank determination strategy.

NAME

dgssco - General sparse solver condition number estimate.

SYNOPSIS

```
SUBROUTINE DGSSCO ( COND, HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  COND
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSCO - Condition number estimate.

PARAMETERS

COND - DOUBLE PRECISION

On exit, an estimate of the condition number of the factored matrix. Must be called after the numerical factorization subroutine, DGSSFA().

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-700 : Invalid calling sequence - need to call DGSSFA first.
-710 : Condition number estimate not available (not implemented
       for this HANDLE's matix type).
```

NAME

dgssda - Deallocate working storage for the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSDA ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSDA - Deallocate dynamically allocated working storage.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

none

NAME

dgssfa - General sparse solver numeric factorization.

SYNOPSIS

```
SUBROUTINE DGSSFA ( NEQNS, COLSTR, ROWIND, VALUES, HANDLE, IER )

INTEGER           NEQNS, COLSTR(*), ROWIND(*), IER
DOUBLE PRECISION  VALUES(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSFA - Numeric factorization of a sparse matrix.

PARAMETERS

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

`COLSTR(*)` - INTEGER array

On entry, `COLSTR(*)` is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

`ROWIND(*)` - INTEGER array

On entry, `ROWIND(*)` is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

`VALUES(*)` - DOUBLE PRECISION array

On entry, `VALUES(*)` is an array of size COLSTR(NEQNS+1)-1, containing the numeric values of the sparse matrix to be factored. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-300 : Invalid calling sequence - need to call DGSSOR first.
-301 : Failure to dynamically allocate memory.
-666 : Internal error.
```

NAME

dgssfs - General sparse solver one call interface.

SYNOPSIS

```
SUBROUTINE DGSSFS ( MTXTYP, PIVOT , NEQNS, COLSTR, ROWIND,
                    VALUES, NRHS  , RHS   , LDRHS , ORDMTHD,
                    OUTUNT, MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), NRHS, LDRHS,
                  OUTUNT, MSGLVL, IER
CHARACTER*3       ORDMTHD
DOUBLE PRECISION  VALUES(*), RHS(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSFS - General sparse solver one call interface.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'sp' or 'SP' - symmetric structure, positive-definite values
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, pivot specifies whether or not pivoting is used in the course of the numeric factorization. The valid options

are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

VALUES(*) - DOUBLE PRECISION array

On entry, VALUES(*) is an array of size COLSTR(NEQNS+1)-1, containing the non-zero numeric values of the sparse matrix to be factored. Unchanged on exit.

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

RHS(*) - DOUBLE PRECISION array

On entry, RHS(LDRHS,NRHS) contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see DGSSUO)
```

Unchanged on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array of containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros is less than NEQNS.
-201 : Failure to dynamically allocate memory.
-301 : Failure to dynamically allocate memory.
-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS
-666 : Internal error.

NAME

dgssin - Initialize the general sparse solver.

SYNOPSIS

```
SUBROUTINE DGSSIN ( MTXTYP, PIVOT, NEQNS, COLSTR, ROWIND, OUTUNT,
                    MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), OUTUNT, MSGLVL, IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSIN - Initialize the sparse solver and input the matrix structure.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'sp' or 'SP' - symmetric structure, positive-definite values
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, PIVOT specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
```

```
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros less than NEQNS.
```

NAME

dgssor - General sparse solver ordering and symbolic factorization.

SYNOPSIS

```
SUBROUTINE DGSSOR ( ORDMTHD, HANDLE, IER )

CHARACTER*3       ORDMTHD
INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSOR - Orders and symbolically factors a sparse matrix.

PARAMETERS

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see DGSSUO)
```

Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-200 : Invalid calling sequence - need to call DGSSIN first.
-201 : Failure to dynamically allocate memory.
-666 : Internal error.

NAME

dgssps - Print general sparse solver statics.

SYNOPSIS

```
SUBROUTINE DGSSPS ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSPS - Print solver statistics.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-800 : Invalid calling sequence - need to call DGSSSL first.
-899 : Printed solver statistics not supported this release.
```

NAME

dgssrp - Return permutation used by the general sparse solver.

SYNOPSIS

```
SUBROUTINE DGSSRP ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSRP - Returns the permutation used by the solver for the fill-reducing ordering.

PARAMETERS

PERM(NEQNS) - INTEGER array

Undefined on entry. PERM(NEQNS) is the permutation array used by the sparse solver for the fill-reducing ordering. Modified on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-600 : Invalid calling sequence - need to call DGSSOR first.
```

NAME

dgsssl - Solve routine for the general sparse solver.

SYNOPSIS

```
SUBROUTINE DGSSSL ( NRHS, RHS, LDRHS, HANDLE, IER )

INTEGER           NRHS, LDRHS, IER
DOUBLE PRECISION  RHS(LDRHS,NRHS)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSSL - Triangular solve of a factored sparse matrix.

PARAMETERS

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

`RHS(LDRHS,*)` - DOUBLE PRECISION array

On entry, `RHS(LDRHS,NRHS)` contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-400 : Invalid calling sequence - need to call DGSSFA first.
```

-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS

NAME

dgssuo - User supplied permutation for ordering used in the general sparse solver.

SYNOPSIS

```
SUBROUTINE DGSSUO ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

DGSSUO - User supplied permutation for ordering. Must be called after `DGSSIN()` (sparse solver initialization) and before `DGSSOR()` (sparse solver ordering).

PARAMETERS

`PERM(NEQNS)` - INTEGER array

On entry, `PERM(NEQNS)` is a permutation array supplied by the user for the fill-reducing ordering. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-500 : Invalid calling sequence - need to call DGSSIN first.
```

NAME

dgtcon - estimate the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF

SYNOPSIS

```
 SUBROUTINE DGTCON( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, IWORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)

 SUBROUTINE DGTCON_64( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, IWORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GTCON( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK

 SUBROUTINE GTCON_64( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
 REAL(8) :: ANORM, RCOND
```

```
REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
```

C INTERFACE

#include <sunperf.h>

void dgtcon(char norm, int n, double *low, double *diag, double *up1, double *up2, int *ipivot, double anorm, double *rcond, int *info);

void dgtcon_64(char norm, long n, double *low, double *diag, double *up1, double *up2, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

dgtcon estimates the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.

- **WORK (workspace)**
  `dimension(2*N)`
- **IWORK2 (workspace)**
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

dgthr - Gathers specified elements from y into x.

SYNOPSIS

```
SUBROUTINE DGTHR(NZ, Y, X, INDX)

DOUBLE PRECISION Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE DGTHR_64(NZ, Y, X, INDX)

DOUBLE PRECISION Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHR([NZ], Y, X, INDX)

```
REAL(8), DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHR_64([NZ], Y, X, INDX)

REAL(8), DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

DGTHR - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input)

Vector in full storage form. Unchanged on exit.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

dgthrz - Gather and zero.

SYNOPSIS

```
SUBROUTINE DGTHRZ(NZ, Y, X, INDX)

DOUBLE PRECISION Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE DGTHRZ_64(NZ, Y, X, INDX)

DOUBLE PRECISION Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHRZ([NZ], Y, X, INDX)

```
REAL(8), DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHRZ_64([NZ], Y, X, INDX)

REAL(8), DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

DGTHRZ - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. The gathered elements of y are set to zero. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
  y(indx(i)) = 0
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input/output)

Vector in full storage form. Gathered elements are set to zero.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

dgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
  SUBROUTINE DGTRFS( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
 *      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
 *      INFO)
  CHARACTER * 1 TRANSA
  INTEGER N, NRHS, LDB, LDX, INFO
  INTEGER IPIVOT(*), WORK2(*)
  DOUBLE PRECISION LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*),
WORK(*)

  SUBROUTINE DGTRFS_64( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
 *      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
 *      INFO)
  CHARACTER * 1 TRANSA
  INTEGER*8 N, NRHS, LDB, LDX, INFO
  INTEGER*8 IPIVOT(*), WORK2(*)
  DOUBLE PRECISION LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*),
WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GTRFS( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF, DIAGF,
 *       UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
 *       [WORK2], [INFO])
  CHARACTER(LEN=1) :: TRANSA
  INTEGER :: N, NRHS, LDB, LDX, INFO
  INTEGER, DIMENSION(:) :: IPIVOT, WORK2
  REAL(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
  REAL(8), DIMENSION(:,:) :: B, X

  SUBROUTINE GTRFS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
 *       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
 *       [WORK2], [INFO])
  CHARACTER(LEN=1) :: TRANSA
  INTEGER(8) :: N, NRHS, LDB, LDX, INFO
  INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
  REAL(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
  REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dgtrfs(char transa, int n, int nrhs, double *low, double *diag, double *up, double *lowf, double *diagf, double *upf1, double *upf2, int *ipivot, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dgtrfs_64(char transa, long n, long nrhs, double *low, double *diag, double *up, double *lowf, double *diagf, double *upf1, double *upf2, long *ipivot, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dgtrfs improves the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The diagonal elements of A.
- **UP (input)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.
- **DIAGF (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UPF2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 <= i <= n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dgtsv - solve the equation A*X = B,

SYNOPSIS

```
SUBROUTINE DGTSV( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION LOW(*), DIAG(*), UP(*), B(LDB,*)

SUBROUTINE DGTSV_64( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION LOW(*), DIAG(*), UP(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GTSV( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: LOW, DIAG, UP
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE GTSV_64( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: LOW, DIAG, UP
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dgtsv(int n, int nrhs, double *low, double *diag, double *up, double *b, int ldb, int *info);

void dgtsv_64(long n, long nrhs, double *low, double *diag, double *up, double *b, long ldb, long *info);

PURPOSE

dgtsv solves the equation

where A is an n by n tridiagonal matrix, by Gaussian elimination with partial pivoting.

Note that the equation A'*X = B may be solved by interchanging the order of the arguments DU and DL.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-2) elements of the second super-diagonal of the upper triangular matrix U from the LU factorization of A, in LOW(1), ..., LOW(n-2).
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of U.
- **UP (input/output)**
  On entry, UP must contain the (n-1) super-diagonal elements of A.

  On exit, UP is overwritten by the (n-1) elements of the first super-diagonal of U.
- **B (input/output)**
  On entry, the N by NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N by NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, U(i,i) is exactly zero, and the solution
has not been computed.  The factorization has not been
completed unless i  = N.
```

NAME

dgtsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B,

---

SYNOPSIS

```
  SUBROUTINE DGTSVX( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
 *      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
 *      WORK, WORK2, INFO)
  CHARACTER * 1 FACT, TRANSA
  INTEGER N, NRHS, LDB, LDX, INFO
  INTEGER IPIVOT(*), WORK2(*)
  DOUBLE PRECISION RCOND
  DOUBLE PRECISION LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*),
WORK(*)

  SUBROUTINE DGTSVX_64( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
 *      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
 *      WORK, WORK2, INFO)
  CHARACTER * 1 FACT, TRANSA
  INTEGER*8 N, NRHS, LDB, LDX, INFO
  INTEGER*8 IPIVOT(*), WORK2(*)
  DOUBLE PRECISION RCOND
  DOUBLE PRECISION LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*),
WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GTSVX( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
 *       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR,
 *       [WORK], [WORK2], [INFO])
  CHARACTER(LEN=1) :: FACT, TRANSA
  INTEGER :: N, NRHS, LDB, LDX, INFO
  INTEGER, DIMENSION(:) :: IPIVOT, WORK2
  REAL(8) :: RCOND
  REAL(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
  REAL(8), DIMENSION(:,:) :: B, X

  SUBROUTINE GTSVX_64( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP,
 *       LOWF, DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR,
 *       BERR, [WORK], [WORK2], [INFO])
  CHARACTER(LEN=1) :: FACT, TRANSA
  INTEGER(8) :: N, NRHS, LDB, LDX, INFO
  INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
  REAL(8) :: RCOND
  REAL(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
  REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dgtsvx(char fact, char transa, int n, int nrhs, double *low, double *diag, double *up, double *lowf, double *diagf, double *upf1, double *upf2, int *ipivot, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dgtsvx_64(char fact, char transa, long n, long nrhs, double *low, double *diag, double *up, double *lowf, double *diagf, double *upf1, double *upf2, long *ipivot, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dgtsvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B, where A is a tridiagonal matrix of order N and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the LU decomposition is used to factor the matrix A as A = L * U, where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

2. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': LOWF, DIAGF, UPF1, UPF2, and IPIVOT contain the factored form of A; LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2 and IPIVOT will not be modified. = 'N': The matrix will be copied to LOWF, DIAGF, and UPF1 and factored.
- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The n diagonal elements of A.
- **UP (input/output)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input/output)**
  If FACT = 'F', then LOWF is an input argument and on entry contains the (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.

  If FACT = 'N', then LOWF is an output argument and on exit contains the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

  If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input/output)**
  If FACT = 'F', then UPF1 is an input argument and on entry contains the (n-1) elements of the first superdiagonal of U.

  If FACT = 'N', then UPF1 is an output argument and on exit contains the (n-1) elements of the first superdiagonal of U.
- **UPF2 (input/output)**
  If FACT = 'F', then UPF2 is an input argument and on entry contains the (n-2) elements of the second superdiagonal of U.

  If FACT = 'N', then UPF2 is an output argument and on exit contains the (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the LU factorization of A as computed by SGTTRF.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the LU factorization of A; row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has not been completed unless i  = N, but the
factor U is exactly singular, so the solution
and error bounds could not be computed.
RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

dgttrf - compute an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges

---

SYNOPSIS

```
SUBROUTINE DGTTRF( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
INTEGER N, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*)

SUBROUTINE DGTTRF_64( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*)
```

F95 INTERFACE

```
SUBROUTINE GTTRF( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2

SUBROUTINE GTTRF_64( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
```

C INTERFACE

#include <sunperf.h>

void dgttrf(int n, double *low, double *diag, double *up1, double *up2, int *ipivot, int *info);

void dgttrf_64(long n, double *low, double *diag, double *up1, double *up2, long *ipivot, long *info);

PURPOSE

dgttrf computes an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

```
A = L * U
```

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

ARGUMENTS

- **N (input)**
  The order of the matrix A.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input/output)**
  On entry, UP1 must contain the (n-1) super-diagonal elements of A.

  On exit, UP1 is overwritten by the (n-1) elements of the first super-diagonal of U.
- **UP2 (output)**
  On exit, UP2 is overwritten by the (n-2) elements of the second super-diagonal of U.
- **IPIVOT (output)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -k, the k-th argument had an illegal value

 > 0:  if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

dgttrs - solve one of the systems of equations A*X = B or A'*X = B,

---

SYNOPSIS

```
 SUBROUTINE DGTTRS( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NRHS, LDB, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)

 SUBROUTINE DGTTRS_64( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NRHS, LDB, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GTTRS( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NRHS, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
 REAL(8), DIMENSION(:,:) :: B

 SUBROUTINE GTTRS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NRHS, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
 REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dgttrs(char transa, int n, int nrhs, double *low, double *diag, double *up1, double *up2, int *ipivot, double *b, int ldb, int *info);

void dgttrs_64(char transa, long n, long nrhs, double *low, double *diag, double *up1, double *up2, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dgttrs solves one of the systems of equations A*X = B or A'*X = B, with a tridiagonal matrix A using the LU factorization computed by SGTTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first super-diagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second super-diagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input/output)**
  On entry, the matrix of right hand side vectors B. On exit, B is overwritten by the solution vectors X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dhgeqz - implement a single-/double-shift version of the QZ method for finding the generalized eigenvalues `w(j)=(ALPHAR(j)` + i*ALPHAI(j))/BETAR(j) of the equation det(A-w(i) B) = 0 In addition, the pair A,B may be reduced to generalized Schur form

---

SYNOPSIS

```
 SUBROUTINE DHGEQZ( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)

 SUBROUTINE DHGEQZ_64( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B,
*      LDB, ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HGEQZ( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE HGEQZ_64( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dhgeqz(char job, char compq, char compz, int n, int ilo, int ihi, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *q, int ldq, double *z, int ldz, int *info);

void dhgeqz_64(char job, char compq, char compz, long n, long ilo, long ihi, double *a, long lda, double *b, long ldb, double *alphar, double

*alphai, double *beta, double *q, long ldq, double *z, long ldz, long *info);

PURPOSE

dhgeqz implements a single-/double-shift version of the QZ method for finding the generalized eigenvalues B is upper triangular, and A is block upper triangular, where the diagonal blocks are either 1-by-1 or 2-by-2, the 2-by-2 blocks having complex generalized eigenvalues (see the description of the argument JOB.)

If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form by applying one orthogonal tranformation (usually called Q) on the left and another (usually called Z) on the right. The 2-by-2 upper-triangular diagonal blocks of B corresponding to 2-by-2 blocks of A will be reduced to positive diagonal matrices. (I.e., if `A(j+1,j)` is non-zero, then B(j+1,j)=B(j,j+1)=0 and `B(j,j)` and `B(j+1,j+1)` will be positive.)

If JOB='E', then at each iteration, the same transformations are computed, but they are only applied to those parts of A and B which are needed to compute ALPHAR, ALPHAI, and BETAR.

If JOB='S' and COMPQ and COMPZ are 'V' or 'I', then the orthogonal transformations used to reduce (A,B) are accumulated into the arrays Q and Z s.t.:

(in) `A(in)` Z(in)* = `Q(out)` `A(out)` Z(out)* (in) `B(in)` Z(in)* = `Q(out)` `B(out)` Z(out)*

Ref: C.B. Moler & G.W. Stewart, ``An Algorithm for Generalized Matrixigenvalue Problems", SIAM J. Numer. Anal., 10(1973),p. 241--256.

ARGUMENTS

- **JOB (input)**

```
 = 'E': compute only ALPHAR, ALPHAI, and BETA.  A and B will
not necessarily be put into generalized Schur form.
 = 'S': put A and B into generalized Schur form, as well
as computing ALPHAR, ALPHAI, and BETA.
```

- **COMPQ (input)**

```
 = 'N': do not modify Q.

 = 'V': multiply the array Q on the right by the transpose of
the orthogonal tranformation that is applied to the
left side of A and B to reduce them to Schur form.
 = 'I': like COMPQ ='V', except that Q will be initialized to
the identity first.
```

- **COMPZ (input)**

```
 = 'N': do not modify Z.

 = 'V': multiply the array Z on the right by the orthogonal
tranformation that is applied to the right side of
A and B to reduce them to Schur form.
 = 'I': like COMPZ ='V', except that Z will be initialized to
the identity first.
```

- **N (input)**
  The order of the matrices A, B, Q, and Z. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**

On entry, the N-by-N upper Hessenberg matrix A. Elements below the subdiagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to generalized Schur form. If JOB ='E', then on exit A will have been destroyed. The diagonal blocks will be correct, but the off-diagonal portion will be meaningless.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. Elements below the diagonal must be zero. 2-by-2 blocks in B corresponding to 2-by-2 blocks in A will be reduced to positive diagonal form. (I.e., if `A(j+1,j)` is non-zero, then `B(j+1,j)` =B(j,j+1) =0 and `B(j,j)` and `B(j+1,j+1)` will be positive.) If JOB ='S', then on exit A and B will have been simultaneously reduced to Schur form. If JOB ='E', then on exit B will have been destroyed. Elements corresponding to diagonal blocks of A will be correct, but the off-diagonal portion will be meaningless.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **ALPHAR (output)**
  `ALPHAR(1:N)` will be set to real parts of the diagonal elements of A that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `ALPHAR(j)` =A(j,j). Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB.
- **ALPHAI (output)**
  `ALPHAI(1:N)` will be set to imaginary parts of the diagonal elements of A that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `ALPHAR(j)` =0. Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB.
- **BETA (output)**
  `BETA(1:N)` will be set to the (real) diagonal elements of B that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `BETA(j)` =B(j,j). Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB. (Note that `BETA(1:N)` will always be non-negative, and no BETAI is necessary.)
- **Q (input/output)**
  If COMPQ ='N', then Q will not be referenced. If COMPQ ='V' or 'I', then the transpose of the orthogonal transformations which are applied to A and B on the left will be applied to the array Q on the right.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If COMPQ ='V' or 'I', then LDQ > = N.
- **Z (input/output)**
  If COMPZ ='N', then Z will not be referenced. If COMPZ ='V' or 'I', then the orthogonal transformations which are applied to A and B on the right will be applied to the array Z on the right.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If COMPZ ='V' or 'I', then LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO > = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1,...,N: the QZ iteration did not converge.  (A,B) is not
in Schur form, but ALPHAR(i), ALPHAI(i), and
BETA(i), i =INFO+1,...,N should be correct.
 = N+1,...,2*N: the shift calculation failed.  (A,B) is not
in Schur form, but ALPHAR(i), ALPHAI(i), and
BETA(i), i =INFO-N+1,...,N should be correct.
 > 2*N:     various "impossible" errors.
```

FURTHER DETAILS

Iteration counters:

JITER -- counts iterations.

IITER -- counts iterations run since ILAST was last

```
          changed.  This is therefore reset only when a 1-by-1 or
          2-by-2 block deflates off the bottom.
```

NAME

dhsein - use inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H

SYNOPSIS

```
 SUBROUTINE DHSEIN( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, WR, WI,
*      VL, LDVL, VR, LDVR, MM, M, WORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 INTEGER N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER IFAILL(*), IFAILR(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION H(LDH,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE DHSEIN_64( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, WR,
*      WI, VL, LDVL, VR, LDVR, MM, M, WORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 INTEGER*8 N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER*8 IFAILL(*), IFAILR(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION H(LDH,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEIN( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], WR,
*       WI, VL, [LDVL], VR, [LDVR], MM, M, [WORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 INTEGER :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER, DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: H, VL, VR

 SUBROUTINE HSEIN_64( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], WR,
*       WI, VL, [LDVL], VR, [LDVR], MM, M, [WORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 INTEGER(8) :: N, LDH, LDVL, LDVR, MM, M, INFO
```

```
INTEGER(8), DIMENSION(:) :: IFAILL, IFAILR
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8), DIMENSION(:) :: WR, WI, WORK
REAL(8), DIMENSION(:,:) :: H, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dhsein(char side, char eigsrc, char initv, logical *select, int n, double *h, int ldh, double *wr, double *wi, double *vl, int ldvl, double *vr, int ldvr, int mm, int *m, int *ifaill, int *ifailr, int *info);

void dhsein_64(char side, char eigsrc, char initv, logical *select, long n, double *h, long ldh, double *wr, double *wi, double *vl, long ldvl, double *vr, long ldvr, long mm, long *m, long *ifaill, long *ifailr, long *info);

PURPOSE

dhsein uses inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H.

The right eigenvector x and the left eigenvector y of the matrix H corresponding to an eigenvalue w are defined by:

```
H * x = w * x,     y**h * H = w * y**h
```

where y**h denotes the conjugate transpose of the vector y.

ARGUMENTS

- **SIDE (input)**

```
 = 'R': compute right eigenvectors only;

 = 'L': compute left eigenvectors only;

 = 'B': compute both right and left eigenvectors.
```

- **EIGSRC (input)**
  Specifies the source of eigenvalues supplied in (WR,WI):

```
 = 'Q': the eigenvalues were found using SHSEQR; thus, if
H has zero subdiagonal elements, and so is
block-triangular, then the j-th eigenvalue can be
assumed to be an eigenvalue of the block containing
the j-th row/column.  This property allows SHSEIN to
perform inverse iteration on just one diagonal block.
 = 'N': no assumptions are made on the correspondence
between eigenvalues and diagonal blocks.  In this
case, SHSEIN must always perform inverse iteration
using the whole matrix H.
```

- **INITV (input)**

= 'N': no initial vectors are supplied;

= 'U': user-supplied initial vectors are stored in the arrays
VL and/or VR.

- **SELECT (input/output)**
  Specifies the eigenvectors to be computed. To select the real eigenvector corresponding to a real eigenvalue WR(j), SELECT(j) must be set to .TRUE.. To select the complex eigenvector corresponding to a complex eigenvalue (WR(j),WI(j)), with complex conjugate (WR(j+1),WI(j+1)), either SELECT(j) or SELECT(j+1) or both must be set to .TRUE.; then on exit SELECT(j) is .TRUE. and SELECT(j+1) is .FALSE..
- **N (input)**
  The order of the matrix H. N >= 0.
- **H (input)**
  The upper Hessenberg matrix H.
- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **WR (input/output)**
  On entry, the real and imaginary parts of the eigenvalues of H; a complex conjugate pair of eigenvalues must be stored in consecutive elements of WR and WI. On exit, WR may have been altered since close eigenvalues are perturbed slightly in searching for independent eigenvectors.
- **WI (input)**
  See the description of WR.
- **VL (input/output)**
  On entry, if INITV = 'U' and SIDE = 'L' or 'B', VL must contain starting vectors for the inverse iteration for the left eigenvectors; the starting vector for each eigenvector must be in the same column(s) in which the eigenvector will be stored. On exit, if SIDE = 'L' or 'B', the left eigenvectors specified by SELECT will be stored consecutively in the columns of VL, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= max(1,N) if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if INITV = 'U' and SIDE = 'R' or 'B', VR must contain starting vectors for the inverse iteration for the right eigenvectors; the starting vector for each eigenvector must be in the same column(s) in which the eigenvector will be stored. On exit, if SIDE = 'R' or 'B', the right eigenvectors specified by SELECT will be stored consecutively in the columns of VR, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= max(1,N) if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR required to store the eigenvectors; each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  dimension((N+2)*N)
- **IFAILL (output)**
  If SIDE = 'L' or 'B', IFAILL(i) = j > 0 if the left eigenvector in the i-th column of VL (corresponding to the eigenvalue w(j)) failed to converge; IFAILL(i) = 0 if the eigenvector converged satisfactorily. If the i-th and (i+1)th columns of VL hold a complex eigenvector, then IFAILL(i) and IFAILL(i+1) are set to the same value. If SIDE = 'R', IFAILL is not referenced.
- **IFAILR (output)**

If SIDE = 'R' or 'B', IFAILR(i) = j > 0 if the right eigenvector in the i-th column of VR (corresponding to the eigenvalue w(j)) failed to converge; IFAILR(i) = 0 if the eigenvector converged satisfactorily. If the i-th and (i+1)th columns of VR hold a complex eigenvector, then IFAILR(i) and IFAILR(i+1) are set to the same value. If SIDE = 'L', IFAILR is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, i is the number of eigenvectors which
failed to converge; see IFAILL and IFAILR for further
details.
```

FURTHER DETAILS

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x|+|y|.

NAME

dhseqr - compute the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors

---

SYNOPSIS

```
 SUBROUTINE DHSEQR( JOB, COMPZ, N, ILO, IHI, H, LDH, WR, WI, Z, LDZ,
*      WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 INTEGER N, ILO, IHI, LDH, LDZ, LWORK, INFO
 DOUBLE PRECISION H(LDH,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DHSEQR_64( JOB, COMPZ, N, ILO, IHI, H, LDH, WR, WI, Z,
*      LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 INTEGER*8 N, ILO, IHI, LDH, LDZ, LWORK, INFO
 DOUBLE PRECISION H(LDH,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEQR( JOB, COMPZ, N, ILO, IHI, H, [LDH], WR, WI, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 INTEGER :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: H, Z

 SUBROUTINE HSEQR_64( JOB, COMPZ, N, ILO, IHI, H, [LDH], WR, WI, Z,
*       [LDZ], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: H, Z
```

C INTERFACE

#include <sunperf.h>

void dhseqr(char job, char compz, int n, int ilo, int ihi, double *h, int ldh, double *wr, double *wi, double *z, int ldz, int *info);

void dhseqr_64(char job, char compz, long n, long ilo, long ihi, double *h, long ldh, double *wr, double *wi, double *z, long ldz, long *info);

PURPOSE

dhseqr computes the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q, so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q: A = Q*H*Q**T = (QZ)*T*(QZ)**T.

ARGUMENTS

- **JOB (input)**

```
= 'E':  compute eigenvalues only;

= 'S':  compute eigenvalues and the Schur form T.
```

- **COMPZ (input)**

```
= 'N':  no Schur vectors are computed;

= 'I':  Z is initialized to the unit matrix and the matrix Z
of Schur vectors of H is returned;
= 'V':  Z must contain an orthogonal matrix Q on entry, and
the product Q*Z is returned.
```

- **N (input)**
  The order of the matrix H. N > = 0.
- **ILO (input)**
  It is assumed that H is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGEBAL, and then passed to SGEHRD when the matrix output by SGEBAL is reduced to Hessenberg form. Otherwise ILO and IHI should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **H (input/output)**
  On entry, the upper Hessenberg matrix H. On exit, if JOB = 'S', H contains the upper quasi-triangular matrix T from the Schur decomposition (the Schur form); 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with `H(i,i)` = `H(i+1,i+1)` and `H(i+1,i)*H(i,i+1)` < 0. If

JOB = 'E', the contents of H are unspecified on exit.

- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **WR (output)**
  The real and imaginary parts, respectively, of the computed eigenvalues. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of WR and WI, say the i-th and (i+1)th, with `WI(i)` > 0 and `WI(i+1)` < 0. If JOB = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in H, with `WR(i)` = `H(i,i)` and, if `H(i:i+1,i:i+1)` is a 2-by-2 diagonal block, `WI(i)` = `sqrt(H(i+1,i)*H(i,i+1))` and `WI(i+1)` = -WI(i).
- **WI (output)**
  See the description of WR.
- **Z (input/output)**
  If COMPZ = 'N': Z is not referenced.

  If COMPZ = 'I': on entry, Z need not be set, and on exit, Z contains the orthogonal matrix Z of the Schur vectors of H. If COMPZ = 'V': on entry Z must contain an N-by-N matrix Q, which is assumed to be equal to the unit matrix except for the submatrix Z(ILO:IHI,ILO:IHI); on exit Z contains Q*Z. Normally Q is the orthogonal matrix generated by SORGHR after the call to SGEHRD which formed the Hessenberg matrix H.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= `max(1,N)` if COMPZ = 'I' or 'V'; LDZ >= 1 otherwise.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, SHSEQR failed to compute all of the
eigenvalues in a total of 30*(IHI-ILO+1) iterations;
elements 1:ilo-1 and i+1:n of WR and WI contain those
eigenvalues which have been successfully computed.
```

NAME

dlagtf - factorize the matrix (T-lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as T-lambda*I = PLU

SYNOPSIS

```
SUBROUTINE DLAGTF( N, A, LAMBDA, B, C, TOL, D, IN, INFO)
INTEGER N, INFO
INTEGER IN(*)
DOUBLE PRECISION LAMBDA, TOL
DOUBLE PRECISION A(*), B(*), C(*), D(*)

SUBROUTINE DLAGTF_64( N, A, LAMBDA, B, C, TOL, D, IN, INFO)
INTEGER*8 N, INFO
INTEGER*8 IN(*)
DOUBLE PRECISION LAMBDA, TOL
DOUBLE PRECISION A(*), B(*), C(*), D(*)
```

F95 INTERFACE

```
SUBROUTINE LAGTF( [N], A, LAMBDA, B, C, TOL, D, IN, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IN
REAL(8) :: LAMBDA, TOL
REAL(8), DIMENSION(:) :: A, B, C, D

SUBROUTINE LAGTF_64( [N], A, LAMBDA, B, C, TOL, D, IN, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IN
REAL(8) :: LAMBDA, TOL
REAL(8), DIMENSION(:) :: A, B, C, D
```

C INTERFACE

#include <sunperf.h>

void dlagtf(int n, double *a, double lambda, double *b, double *c, double tol, double *d, int *in, int *info);

void dlagtf_64(long n, double *a, double lambda, double *b, double *c, double tol, double *d, long *in, long *info);

PURPOSE

dlagtf factorizes the matrix (T - lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

The parameter LAMBDA is included in the routine so that SLAGTF may be used, in conjunction with SLAGTS, to obtain eigenvectors of T by inverse iteration.

ARGUMENTS

- **N (input)**
  The order of the matrix T.
- **A (input/output)**
  On entry, A must contain the diagonal elements of T.

  On exit, A is overwritten by the n diagonal elements of the upper triangular matrix U of the factorization of T.
- **LAMBDA (input)**
  On entry, the scalar lambda.
- **B (input/output)**
  On entry, B must contain the (n-1) super-diagonal elements of T.

  On exit, B is overwritten by the (n-1) super-diagonal elements of the matrix U of the factorization of T.
- **C (input/output)**
  On entry, C must contain the (n-1) sub-diagonal elements of T.

  On exit, C is overwritten by the (n-1) sub-diagonal elements of the matrix L of the factorization of T.
- **TOL (input)**
  On entry, a relative tolerance used to indicate whether or not the matrix (T - lambda*I) is nearly singular. TOL should normally be chose as approximately the largest relative error in the elements of T. For example, if the elements of T are correct to about 4 significant figures, then TOL should be set to about 5*10**(-4). If TOL is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of TOL.
- **D (output)**
  On exit, D is overwritten by the (n-2) second super-diagonal elements of the matrix U of the factorization of T.
- **IN (output)**
  On exit, IN contains details of the permutation matrix P. If an interchange occurred at the kth step of the elimination, then `IN(k)` = 1, otherwise `IN(k)` = 0. The element `IN(n)` returns the smallest positive integer j such that

  abs(`u(j,j)`).le. norm((T - lambda*I)(j))*TOL,

where norm(A(j)) denotes the sum of the absolute values of the jth row of the matrix A. If no such j exists then IN(n) is returned as zero. If IN(n) is returned as positive, then a diagonal element of U is small, indicating that (T - lambda*I) is singular or nearly singular,

- **INFO (output)**

```
= 0   : successful exit
```

.lt. 0: if INFO = -k, the kth argument had an illegal value

NAME

dlamrg - will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order

---

SYNOPSIS

```
SUBROUTINE DLAMRG( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER N1, N2, TRD1, TRD2
INTEGER INDEX(*)
DOUBLE PRECISION A(*)

SUBROUTINE DLAMRG_64( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER*8 N1, N2, TRD1, TRD2
INTEGER*8 INDEX(*)
DOUBLE PRECISION A(*)
```

F95 INTERFACE

```
SUBROUTINE LAMRG( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER :: N1, N2, TRD1, TRD2
INTEGER, DIMENSION(:) :: INDEX
REAL(8), DIMENSION(:) :: A

SUBROUTINE LAMRG_64( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER(8) :: N1, N2, TRD1, TRD2
INTEGER(8), DIMENSION(:) :: INDEX
REAL(8), DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void dlamrg(int n1, int n2, double *a, int trd1, int trd2, int *index);

void dlamrg_64(long n1, long n2, double *a, long trd1, long trd2, long *index);

PURPOSE

dlamrg will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

ARGUMENTS

- **N1 (input)**
  Length of the first sequence to be merged.
- **N2 (input)**
  Length of the second sequence to be merged.
- **A (input)**
  On entry, the first N1 elements of A contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final N2 elements.
- **TRD1 (input)**
  Describes the stride to be taken through the array A for the first N1 elements.

  ```
  = -1 subset is sorted in descending order.

  = 1 subset is sorted in ascending order.
  ```
- **TRD2 (input)**
  Describes the stride to be taken through the array A for the first N1 elements.

  ```
  = -1 subset is sorted in descending order.

  = 1 subset is sorted in ascending order.
  ```
- **INDEX (output)**
  On exit this array will contain a permutation such that if B(I) = A(INDEX(I)) for I =1,N1+N2, then B will be sorted in ascending order.

NAME

dlarz - applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right

---

SYNOPSIS

```
SUBROUTINE DLARZ( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER M, N, L, INCV, LDC
DOUBLE PRECISION TAU
DOUBLE PRECISION V(*), C(LDC,*), WORK(*)

SUBROUTINE DLARZ_64( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER*8 M, N, L, INCV, LDC
DOUBLE PRECISION TAU
DOUBLE PRECISION V(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE LARZ( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC], [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER :: M, N, L, INCV, LDC
REAL(8) :: TAU
REAL(8), DIMENSION(:) :: V, WORK
REAL(8), DIMENSION(:,:) :: C

SUBROUTINE LARZ_64( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER(8) :: M, N, L, INCV, LDC
REAL(8) :: TAU
REAL(8), DIMENSION(:) :: V, WORK
REAL(8), DIMENSION(:,:) :: C
```

C INTERFACE

#include <sunperf.h>

void dlarz(char side, int m, int n, int l, double *v, int incv, double tau, double *c, int ldc);

void dlarz_64(char side, long m, long n, long l, double *v, long incv, double tau, double *c, long ldc);

PURPOSE

dlarz applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right. H is represented in the form

```
H = I - tau * v * v'
```

where tau is a real scalar and v is a real vector.

If tau = 0, then H is taken to be the unit matrix.

H is a product of k elementary reflectors as returned by STZRZF.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form  H * C

  = 'R': form  C * H
  ```
- **M (input)**
  The number of rows of the matrix C.
- **N (input)**
  The number of columns of the matrix C.
- **L (input)**
  The number of entries of the vector V containing the meaningful part of the Householder vectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  The vector v in the representation of H as returned by STZRZF. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0.
- **TAU (input)**
  The value tau in the representation of H.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by the matrix H * C if SIDE = 'L', or C * H if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' or (M) if SIDE = 'R'

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA

NAME

dlarzb - applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right

---

SYNOPSIS

```
 SUBROUTINE DLARZB( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V, LDV,
*      T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 INTEGER M, N, K, L, LDV, LDT, LDC, LDWORK
 DOUBLE PRECISION V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)

 SUBROUTINE DLARZB_64( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V,
*      LDV, T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 INTEGER*8 M, N, K, L, LDV, LDT, LDC, LDWORK
 DOUBLE PRECISION V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
```

F95 INTERFACE

```
 SUBROUTINE LARZB( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 INTEGER :: M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL(8), DIMENSION(:,:) :: V, T, C, WORK

 SUBROUTINE LARZB_64( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 INTEGER(8) :: M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL(8), DIMENSION(:,:) :: V, T, C, WORK
```

C INTERFACE

#include <sunperf.h>

void dlarzb(char side, char trans, char direct, char storev, int m, int n, int k, int l, double *v, int ldv, double *t, int ldt, double *c, int ldc, int ldwork);

void dlarzb_64(char side, char trans, char direct, char storev, long m, long n, long k, long l, double *v, long ldv, double *t, long ldt, double *c, long ldc, long ldwork);

PURPOSE

dlarzb applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right.

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **SIDE (input)**

  = 'L': apply H or H' from the Left

  = 'R': apply H or H' from the Right

- **TRANS (input)**

  = 'N': apply H (No transpose)

  = 'C': apply H' (Transpose)

- **DIRECT (input)**
  Indicates how H is formed from a product of elementary reflectors = 'F': H = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H = H(k) . . . H(2) H(1) (Backward)

- **STOREV (input)**
  Indicates how the vectors which define the elementary reflectors are stored:

  = 'C': Columnwise (not supported yet)

  = 'R': Rowwise

- **M (input)**
  The number of rows of the matrix C.

- **N (input)**
  The number of columns of the matrix C.

- **K (input)**
  The order of the matrix T (= the number of elementary reflectors whose product defines the block reflector).

- **L (input)**

The number of columns of the matrix V containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.

- **V (input)**
  If STOREV = 'C', NV = K; if STOREV = 'R', NV = L.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = L; if STOREV = 'R', LDV > = K.
- **T (input)**
  The triangular K-by-K matrix T in the representation of the block reflector.
- **LDT (input)**
  The leading dimension of the array T. LDT > = K.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by H*C or H'*C or C*H or C*H'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  `dimension(MAX(M,N),K)`
- **LDWORK (input)**
  The leading dimension of the array WORK. If SIDE = 'L', LDWORK > = max(1,N); if SIDE = 'R', LDWORK > = max(1,M).

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

dlarzt - form the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors

---

SYNOPSIS

```
SUBROUTINE DLARZT( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
INTEGER N, K, LDV, LDT
DOUBLE PRECISION V(LDV,*), TAU(*), T(LDT,*)

SUBROUTINE DLARZT_64( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
INTEGER*8 N, K, LDV, LDT
DOUBLE PRECISION V(LDV,*), TAU(*), T(LDT,*)
```

F95 INTERFACE

```
SUBROUTINE LARZT( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
INTEGER :: N, K, LDV, LDT
REAL(8), DIMENSION(:) :: TAU
REAL(8), DIMENSION(:,:) :: V, T

SUBROUTINE LARZT_64( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
INTEGER(8) :: N, K, LDV, LDT
REAL(8), DIMENSION(:) :: TAU
REAL(8), DIMENSION(:,:) :: V, T
```

C INTERFACE

#include <sunperf.h>

void dlarzt(char direct, char storev, int n, int k, double *v, int ldv, double *tau, double *t, int ldt);

void dlarzt_64(char direct, char storev, long n, long k, double *v, long ldv, double *tau, double *t, long ldt);

PURPOSE

dlarzt forms the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors.

If DIRECT = 'F', H = `H(1) H(2) . . . H(k)` and T is upper triangular;

If DIRECT = 'B', H = `H(k) . . . H(2) H(1)` and T is lower triangular.

If STOREV = 'C', the vector which defines the elementary reflector `H(i)` is stored in the i-th column of the array V, and

```
H  =  I - V * T * V'
```

If STOREV = 'R', the vector which defines the elementary reflector `H(i)` is stored in the i-th row of the array V, and

```
H  =  I - V' * T * V
```

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **DIRECT (input)**
  Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

  ```
  = 'F': H  = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H  = H(k) . . . H(2) H(1) (Backward)
  ```
- **STOREV (input)**
  Specifies how the vectors which define the elementary reflectors are stored (see also Further Details):

  ```
  = 'R': rowwise
  ```
- **N (input)**
  The order of the block reflector H. N > = 0.
- **K (input)**
  The order of the triangular factor T (= the number of elementary reflectors). K > = 1.
- **V (input)**
  (LDV,K) if STOREV = 'C' (LDV,N) if STOREV = 'R' The matrix V. See further details.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = max(1,N); if STOREV = 'R', LDV > = K.

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i).
- **T (output)**
  The k by k triangular factor T of the block reflector. If DIRECT = 'F', T is upper triangular; if DIRECT = 'B', T is lower triangular. The rest of the array is not used.
- **LDT (input)**
  The leading dimension of the array T. LDT >= K.

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The shape of the matrix V and the storage of the vectors which define the H(i) is best illustrated by the following example with n = 5 and k = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

DIRECT = 'F' and STOREV = 'C': DIRECT = 'F' and STOREV = 'R':

```
                                              ______V_____

     ( v1 v2 v3 )                            /           \
     ( v1 v2 v3 )                          ( v1 v1 v1 v1 v1 . . . . 1 )
V  = ( v1 v2 v3 )                           ( v2 v2 v2 v2 v2 . . . 1   )
     ( v1 v2 v3 )                          ( v3 v3 v3 v3 v3 . . 1     )
     ( v1 v2 v3 )

        .  .  .

        .  .  .

        1  .  .

           1  .

              1
```

DIRECT = 'B' and STOREV = 'C': DIRECT = 'B' and STOREV = 'R':

```
                                                    ______V_____
        1                                          /            \
        .  1                                    ( 1 . . . . v1 v1 v1 v1 v1 )
        .  .  1                                 ( . 1 . . . v2 v2 v2 v2 v2 )
        .  .  .                                 ( . . 1 . . v3 v3 v3 v3 v3 )
        .  .  .

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

```
V  = ( v1 v2 v3 )

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

NAME

dlasrt - the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D')

SYNOPSIS

```
SUBROUTINE DLASRT( ID, N, D, INFO)
CHARACTER * 1 ID
INTEGER N, INFO
DOUBLE PRECISION D(*)

SUBROUTINE DLASRT_64( ID, N, D, INFO)
CHARACTER * 1 ID
INTEGER*8 N, INFO
DOUBLE PRECISION D(*)
```

F95 INTERFACE

```
SUBROUTINE LASRT( ID, [N], D, [INFO])
CHARACTER(LEN=1) :: ID
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: D

SUBROUTINE LASRT_64( ID, [N], D, [INFO])
CHARACTER(LEN=1) :: ID
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: D
```

C INTERFACE

#include <sunperf.h>

void dlasrt(char id, int n, double *d, int *info);

void dlasrt_64(char id, long n, double *d, long *info);

PURPOSE

dlasrt the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D').

Use Quick Sort, reverting to Insertion sort on arrays of

size <= 20. Dimension of STACK limits N to about 2**32.

ARGUMENTS

- **ID (input)**

```
= 'I': sort D in increasing order;

= 'D': sort D in decreasing order.
```

- **N (input)**
  The length of the array D.
- **D (input/output)**
  On entry, the array to be sorted. On exit, D has been sorted into increasing order (D(1) < = ... < = `D(N)`) or into decreasing order (D(1) > = ... > = `D(N)`), depending on ID.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dlatzm - routine is deprecated and has been replaced by routine SORMRZ

SYNOPSIS

```
SUBROUTINE DLATZM( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER M, N, INCV, LDC
DOUBLE PRECISION TAU
DOUBLE PRECISION V(*), C1(LDC,*), C2(LDC,*), WORK(*)

SUBROUTINE DLATZM_64( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER*8 M, N, INCV, LDC
DOUBLE PRECISION TAU
DOUBLE PRECISION V(*), C1(LDC,*), C2(LDC,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE LATZM( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER :: M, N, INCV, LDC
REAL(8) :: TAU
REAL(8), DIMENSION(:) :: V, WORK
REAL(8), DIMENSION(:,:) :: C1, C2

SUBROUTINE LATZM_64( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER(8) :: M, N, INCV, LDC
REAL(8) :: TAU
REAL(8), DIMENSION(:) :: V, WORK
REAL(8), DIMENSION(:,:) :: C1, C2
```

C INTERFACE

#include <sunperf.h>

void dlatzm(char side, int m, int n, double *v, int incv, double tau, double *c1, double *c2, int ldc);

void dlatzm_64(char side, long m, long n, double *v, long incv, double tau, double *c1, double *c2, long ldc);

PURPOSE

dlatzm routine is deprecated and has been replaced by routine SORMRZ.

SLATZM applies a Householder matrix generated by STZRQF to a matrix.

Let P = I - tau*u*u', u = (1),

```
                                ( v )
```

where v is an (m-1) vector if SIDE = 'L', or a (n-1) vector if SIDE = 'R'.

If SIDE equals 'L', let

```
       C = [ C1 ] 1
           [ C2 ] m-1
             n
```

Then C is overwritten by P*C.

If SIDE equals 'R', let

```
       C = [ C1, C2 ] m
             1  n-1
```

Then C is overwritten by C*P.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form P * C

  = 'R': form C * P
  ```

- **M (input)**
  The number of rows of the matrix C.
- **N (input)**

The number of columns of the matrix C.

- **V (input)**
  (1 + (M-1)*abs(INCV)) if SIDE = 'L' (1 + (N-1)*abs(INCV)) if SIDE = 'R' The vector v in the representation of P. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0
- **TAU (input)**
  The value tau in the representation of P.
- **C1 (input/output)**
  (LDC,N) if SIDE = 'L' (M,1) if SIDE = 'R' On entry, the n-vector C1 if SIDE = 'L', or the m-vector C1 if SIDE = 'R'.

  On exit, the first row of P*C if SIDE = 'L', or the first column of C*P if SIDE = 'R'.
- **C2 (input/output)**
  (LDC, N) if SIDE = 'L' (LDC, N-1) if SIDE = 'R' On entry, the (m - 1) x n matrix C2 if SIDE = 'L', or the m x (n - 1) matrix C2 if SIDE = 'R'.

  On exit, rows 2:m of P*C if SIDE = 'L', or columns 2:m of C*P if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the arrays C1 and C2. LDC > = (1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'

NAME

dnrm2 - Return the Euclidian norm of a vector.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DNRM2( N, X, INCX)
INTEGER N, INCX
DOUBLE PRECISION X(*)

DOUBLE PRECISION FUNCTION DNRM2_64( N, X, INCX)
INTEGER*8 N, INCX
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION NRM2( [N], X, [INCX])
INTEGER :: N, INCX
REAL(8), DIMENSION(:) :: X

REAL(8) FUNCTION NRM2_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

double dnrm2(int n, double *x, int incx);

double dnrm2_64(long n, double *x, long incx);

PURPOSE

dnrm2 Return the Euclidian norm of a vector x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NAME

dopgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by SSPTRD using packed storage

---

SYNOPSIS

```
SUBROUTINE DOPGTR( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDQ, INFO
DOUBLE PRECISION AP(*), TAU(*), Q(LDQ,*), WORK(*)

SUBROUTINE DOPGTR_64( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDQ, INFO
DOUBLE PRECISION AP(*), TAU(*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE OPGTR( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDQ, INFO
REAL(8), DIMENSION(:) :: AP, TAU, WORK
REAL(8), DIMENSION(:,:) :: Q

SUBROUTINE OPGTR_64( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDQ, INFO
REAL(8), DIMENSION(:) :: AP, TAU, WORK
REAL(8), DIMENSION(:,:) :: Q
```

C INTERFACE

#include <sunperf.h>

void dopgtr(char uplo, int n, double *ap, double *tau, double *q, int ldq, int *info);

void dopgtr_64(char uplo, long n, double *ap, double *tau, double *q, long ldq, long *info);

PURPOSE

dopgtr generates a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by SSPTRD using packed storage:

if UPLO = 'U', Q = `H(n-1)` . . . `H(2)` H(1),

if UPLO = 'L', Q = `H(1)` `H(2)` . . . H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangular packed storage used in previous
call to SSPTRD;
 = 'L': Lower triangular packed storage used in previous
call to SSPTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **AP (input)**
  The vectors which define the elementary reflectors, as returned by SSPTRD.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSPTRD.
- **Q (output)**
  The N-by-N orthogonal matrix Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **WORK (workspace)**
  `dimension(N-1)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dopmtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DOPMTR( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER M, N, LDC, INFO
 DOUBLE PRECISION AP(*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DOPMTR_64( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER*8 M, N, LDC, INFO
 DOUBLE PRECISION AP(*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE OPMTR( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER :: M, N, LDC, INFO
 REAL(8), DIMENSION(:) :: AP, TAU, WORK
 REAL(8), DIMENSION(:,:) :: C

 SUBROUTINE OPMTR_64( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER(8) :: M, N, LDC, INFO
 REAL(8), DIMENSION(:) :: AP, TAU, WORK
 REAL(8), DIMENSION(:,:) :: C
```

C INTERFACE

#include <sunperf.h>

void dopmtr(char side, char uplo, char trans, int m, int n, double *ap, double *tau, double *c, int ldc, int *info);

void dopmtr_64(char side, char uplo, char trans, long m, long n, double *ap, double *tau, double *c, long ldc, long *info);

PURPOSE

dopmtr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by SSPTRD using packed storage:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
```

- **UPLO (input)**

```
  = 'U': Upper triangular packed storage used in previous
call to SSPTRD;
  = 'L': Lower triangular packed storage used in previous
call to SSPTRD.
```

- **TRANS (input)**

```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **AP (input)**
  (M*(M+1)/2) if SIDE = 'L' (N*(N+1)/2) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SSPTRD. AP is modified by the routine but restored on exit.
- **TAU (input)**
  or (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSPTRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dorg2l - generate an m by n real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE DORG2L( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORG2L_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORG2L( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORG2L_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorg2l(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorg2l_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorg2l L generates an m by n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorg2r - generate an m by n real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE DORG2R( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORG2R_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORG2R( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORG2R_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorg2r(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorg2r_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorg2r R generates an m by n real matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgbr - generate one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form

---

SYNOPSIS

```
SUBROUTINE DORGBR( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
INTEGER M, N, K, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGBR_64( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
INTEGER*8 M, N, K, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGBR( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
INTEGER :: M, N, K, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGBR_64( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgbr(char vect, int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgbr_64(char vect, long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgbr generates one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form: A = Q * B * P**T. Q and P**T are defined as products of elementary reflectors `H(i)` or `G(i)` respectively.

If VECT = 'Q', A is assumed to have been an M-by-K matrix, and Q is of order M:

if m >= k, Q = `H(1) H(2) . . . H(k)` and SORGBR returns the first n columns of Q, where m >= n >= k;

if m < k, Q = `H(1) H(2) . . . H(m-1)` and SORGBR returns Q as an M-by-M matrix.

If VECT = 'P', A is assumed to have been a K-by-N matrix, and P**T is of order N:

if k < n, P**T = `G(k) . . . G(2) G(1)` and SORGBR returns the first m rows of P**T, where n >= m >= k;

if k >= n, P**T = `G(n-1) . . . G(2) G(1)` and SORGBR returns P**T as an N-by-N matrix.

ARGUMENTS

- **VECT (input)**
  Specifies whether the matrix Q or the matrix P**T is required, as defined in the transformation applied by SGEBRD:

  ```
  = 'Q':  generate Q;

  = 'P':  generate P**T.
  ```
- **M (input)**
  The number of rows of the matrix Q or P**T to be returned. M > = 0.
- **N (input)**
  The number of columns of the matrix Q or P**T to be returned. N > = 0. If VECT = 'Q', M > = N > = min(M,K); if VECT = 'P', N > = M > = min(N,K).
- **K (input)**
  If VECT = 'Q', the number of columns in the original M-by-K matrix reduced by SGEBRD. If VECT = 'P', the number of rows in the original K-by-N matrix reduced by SGEBRD. K > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SGEBRD. On exit, the M-by-N matrix Q or P**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).

- **TAU (input)**
  (min(M,K)) if VECT = 'Q' (min(N,K)) if VECT = 'P' `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or G(i), which determines Q or P**T, as returned by SGEBRD in its array argument TAUQ or TAUP.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,min(M,N)). For optimum performance LWORK >= min(M,N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dorghr - generate a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD

SYNOPSIS

```
SUBROUTINE DORGHR( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER N, ILO, IHI, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGHR_64( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER*8 N, ILO, IHI, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGHR( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
INTEGER :: N, ILO, IHI, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGHR_64( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
INTEGER(8) :: N, ILO, IHI, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorghr(int n, int ilo, int ihi, double *a, int lda, double *tau, int *info);

void dorghr_64(long n, long ilo, long ihi, double *a, long lda, double *tau, long *info);

PURPOSE

dorghr generates a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **N (input)**
  The order of the matrix Q. N > = 0.
- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of SGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SGEHRD. On exit, the N-by-N orthogonal matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEHRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = IHI-ILO. For optimum performance LWORK > = (IHI-ILO)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dorgl2 - generate an m by n real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE DORGL2( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGL2_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGL2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGL2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgl2(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgl2_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgl2 generates an m by n real matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorglq - generate an M-by-N real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE DORGLQ( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGLQ_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGLQ( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGLQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorglq(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorglq_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorglq generates an M-by-N real matrix Q with orthonormal rows, which is defined as the first M rows of a product of K elementary reflectors of order N

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. N > = M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M > = K > = 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,M). For optimum performance LDWORK > = M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgql - generate an M-by-N real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE DORGQL( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGQL_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGQL( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGQL_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgql(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgql_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgql generates an M-by-N real matrix Q with orthonormal columns, which is defined as the last N columns of a product of K elementary reflectors of order M

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimum performance LDWORK > = N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgqr - generate an M-by-N real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE DORGQR( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGQR_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGQR( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGQR_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgqr(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgqr_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgqr generates an M-by-N real matrix Q with orthonormal columns, which is defined as the first N columns of a product of K elementary reflectors of order M

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimum performance LDWORK > = N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgr2 - generate an m by n real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE DORGR2( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGR2_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGR2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGR2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgr2(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgr2_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgr2 generates an m by n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. N > = M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M > = K > = 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgrq - generate an M-by-N real matrix Q with orthonormal rows,

---

SYNOPSIS

```
  SUBROUTINE DORGRQ( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
  INTEGER M, N, K, LDA, LDWORK, INFO
  DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

  SUBROUTINE DORGRQ_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
  INTEGER*8 M, N, K, LDA, LDWORK, INFO
  DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE ORGRQ( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
 *        [INFO])
  INTEGER :: M, N, K, LDA, LDWORK, INFO
  REAL(8), DIMENSION(:) :: TAU, WORK
  REAL(8), DIMENSION(:,:) :: A

  SUBROUTINE ORGRQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
 *        [INFO])
  INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
  REAL(8), DIMENSION(:) :: TAU, WORK
  REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgrq(int m, int n, int k, double *a, int lda, double *tau, int *info);

void dorgrq_64(long m, long n, long k, double *a, long lda, double *tau, long *info);

PURPOSE

dorgrq generates an M-by-N real matrix Q with orthonormal rows, which is defined as the last M rows of a product of K elementary reflectors of order N

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

dorgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD

SYNOPSIS

```
SUBROUTINE DORGTR( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DORGTR_64( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGTR( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE ORGTR_64( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dorgtr(char uplo, int n, double *a, int lda, double *tau, int *info);

void dorgtr_64(char uplo, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dorgtr generates a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD:

if UPLO = 'U', Q = `H(n-1) . . . H(2)` H(1),

if UPLO = 'L', Q = `H(1) H(2) . . .` H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from SSYTRD;
 = 'L': Lower triangle of A contains elementary reflectors
from SSYTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SSYTRD. On exit, the N-by-N orthogonal matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSYTRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N-1). For optimum performance LWORK > = (N-1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormbr - VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMBR( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMBR_64( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMBR( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMBR_64( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormbr(char vect, char side, char trans, int m, int n, int k, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormbr_64(char vect, char side, char trans, long m, long n, long k, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormbr VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N': Q * C C * Q TRANS = 'T': Q**T * C C * Q**T

If VECT = 'P', SORMBR overwrites the general real M-by-N matrix C with

```
                SIDE = 'L'     SIDE = 'R'
```

TRANS = 'N': P * C C * P

TRANS = 'T': P**T * C C * P**T

Here Q and P**T are the orthogonal matrices determined by SGEBRD when reducing a real matrix A to bidiagonal form: A = Q * B * P**T. Q and P**T are defined as products of elementary reflectors `H(i)` and `G(i)` respectively.

Let nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Thus nq is the order of the orthogonal matrix Q or P**T that is applied.

If VECT = 'Q', A is assumed to have been an NQ-by-K matrix: if nq >= k, Q = `H(1)` `H(2)` . . . H(k);

if nq < k, Q = `H(1)` `H(2)` . . . H(nq-1).

If VECT = 'P', A is assumed to have been a K-by-NQ matrix: if k < nq, P = `G(1)` `G(2)` . . . G(k);

if k >= nq, P = `G(1)` `G(2)` . . . G(nq-1).

ARGUMENTS

- **VECT (input)**

  ```
  = 'Q': apply Q or Q**T;

  = 'P': apply P or P**T.
  ```

- **SIDE (input)**

  ```
  = 'L': apply Q, Q**T, P or P**T from the Left;

  = 'R': apply Q, Q**T, P or P**T from the Right.
  ```

- **TRANS (input)**

```
= 'N':  No transpose, apply Q  or P;

= 'T':  Transpose, apply Q**T or P**T.
```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  If VECT = 'Q', the number of columns in the original matrix reduced by SGEBRD. If VECT = 'P', the number of rows in the original matrix reduced by SGEBRD. K >= 0.
- **A (input)**
  (LDA,min(nq,K)) if VECT = 'Q' (LDA,nq) if VECT = 'P' The vectors which define the elementary reflectors `H(i)` and G(i), whose products determine the matrices Q and P, as returned by SGEBRD.
- **LDA (input)**
  The leading dimension of the array A. If VECT = 'Q', LDA >= max(1,nq); if VECT = 'P', LDA >= max(1,min(nq,K)).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or `G(i)` which determines Q or P, as returned by SGEBRD in the array argument TAUQ or TAUP.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q or P*C or P**T*C or C*P or C*P**T.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormhr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMHR( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMHR_64( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMHR( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMHR_64( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormhr(char side, char trans, int m, int n, int ilo, int ihi, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormhr_64(char side, char trans, long m, long n, long ilo, long ihi, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormhr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of IHI-ILO elementary reflectors, as returned by SGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of SGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). If SIDE = 'L', then 1 < = ILO < = IHI < = M, if M > 0, and ILO = 1 and IHI = 0, if M = 0; if SIDE = 'R', then 1 < = ILO < = IHI < = N, if N > 0, and ILO = 1 and IHI = 0, if N = 0.

- **IHI (input)**
  See the description of ILO.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SGEHRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**

(M-1) if SIDE = 'L' (N-1) if SIDE = 'R' TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGEHRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormlq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMLQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMLQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMLQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMLQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormlq(char side, char trans, int m, int n, int k, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormlq_64(char side, char trans, long m, long n, long k, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormlq overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by SGELQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M >= K >= 0; if SIDE = 'R', N >= K >= 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,K).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormql - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMQL( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMQL_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMQL( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMQL_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormql(char side, char trans, int m, int n, int k, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormql_64(char side, char trans, long m, long n, long k, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormql overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by SGEQLF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M >= K >= 0; if SIDE = 'R', N >= K >= 0.
- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA >= max(1,M); if SIDE = 'R', LDA >= max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **C (input/output)**

On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormqr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMQR( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
  SIDE

 SUBROUTINE DORMQR_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
  SIDE
```

F95 INTERFACE

```
 SUBROUTINE ORMQR( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
  :: SIDE

 SUBROUTINE ORMQR_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
  :: SIDE
```

C INTERFACE

#include <sunperf.h>

void dormqr(side, char trans, int m, int n, int k, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormqr_64(side, char trans, long m, long n, long k, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormqr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by SGEQRF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormrq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMRQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMRQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMRQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMRQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormrq(char side, char trans, int m, int n, int k, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormrq_64(char side, char trans, long m, long n, long k, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormrq overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by SGERQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M >= K >= 0; if SIDE = 'R', N >= K >= 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,K).
- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dormrz - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, L, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, L, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMRZ( SIDE, TRANS, [M], [N], K, L, A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, L, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMRZ_64( SIDE, TRANS, [M], [N], K, L, A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, L, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormrz(char side, char trans, int m, int n, int k, int l, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormrz_64(char side, char trans, long m, long n, long k, long l, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormrz overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by STZRZF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **L (input)**
  The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by STZRZF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by STZRZF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

dormtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE DORMTR( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER M, N, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE DORMTR_64( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER*8 M, N, LDA, LDC, LWORK, INFO
 DOUBLE PRECISION A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMTR( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER :: M, N, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE ORMTR_64( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER(8) :: M, N, LDA, LDC, LWORK, INFO
 REAL(8), DIMENSION(:) :: TAU, WORK
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dormtr(char side, char uplo, char trans, int m, int n, double *a, int lda, double *tau, double *c, int ldc, int *info);

void dormtr_64(char side, char uplo, char trans, long m, long n, double *a, long lda, double *tau, double *c, long ldc, long *info);

PURPOSE

dormtr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by SSYTRD:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
 = 'L': apply Q or Q**T from the Left;

 = 'R': apply Q or Q**T from the Right.
```

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from SSYTRD;
 = 'L': Lower triangle of A contains elementary reflectors
from SSYTRD.
```

- **TRANS (input)**

```
 = 'N':  No transpose, apply Q;

 = 'T':  Transpose, apply Q**T.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SSYTRD.
- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**
  (M-1) if SIDE = 'L' (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSYTRD.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

SYNOPSIS

```
 SUBROUTINE DPBCON( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DPBCON_64( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBCON( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE PBCON_64( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
```

```
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpbcon(char uplo, int n, int ndiag, double *a, int lda, double anorm, double *rcond, int *info);

void dpbcon_64(char uplo, long n, long ndiag, double *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

dpbcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  `= 'U': Upper triangular factor stored in A;`

  `= 'L': Lower triangular factor stored in A.`
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric band matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**

dimension(N)

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

dpbequ - compute row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
 SUBROUTINE DPBEQU( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INFO
 DOUBLE PRECISION SCOND, AMAX
 DOUBLE PRECISION A(LDA,*), SCALE(*)

 SUBROUTINE DPBEQU_64( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INFO
 DOUBLE PRECISION SCOND, AMAX
 DOUBLE PRECISION A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE PBEQU( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INFO
 REAL(8) :: SCOND, AMAX
 REAL(8), DIMENSION(:) :: SCALE
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE PBEQU_64( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INFO
 REAL(8) :: SCOND, AMAX
 REAL(8), DIMENSION(:) :: SCALE
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpbequ(char uplo, int n, int ndiag, double *a, int lda, double *scale, double *scond, double *amax, int *info);

void dpbequ_64(char uplo, long n, long ndiag, double *a, long lda, double *scale, double *scond, double *amax, long *info);

PURPOSE

dpbequ computes row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular of A is stored;

= 'L':  Lower triangular of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

> 0: if INFO = i, the i-th diagonal element is nonpositive.

NAME

dpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DPBRFS( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPBRFS_64( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBRFS( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE PBRFS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dpbrfs(char uplo, int n, int ndiag, int nrhs, double *a, int lda, double *af, int ldaf, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dpbrfs_64(char uplo, long n, long ndiag, long nrhs, double *a, long lda, double *af, long ldaf, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dpbrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A as computed by SPBTRF, in the same storage format as A (see A).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPBTRS. On exit, the improved solution matrix X.
- **LDX (input)**

The leading dimension of the array X. LDX >= max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dpbstf - compute a split Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE DPBSTF( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER N, KD, LDAB, INFO
DOUBLE PRECISION AB(LDAB,*)

SUBROUTINE DPBSTF_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, KD, LDAB, INFO
DOUBLE PRECISION AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE PBSTF( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, KD, LDAB, INFO
REAL(8), DIMENSION(:,:) :: AB

SUBROUTINE PBSTF_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, KD, LDAB, INFO
REAL(8), DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void dpbstf(char uplo, int n, int kd, double *ab, int ldab, int *info);

void dpbstf_64(char uplo, long n, long kd, double *ab, long ldab, long *info);

PURPOSE

dpbstf computes a split Cholesky factorization of a real symmetric positive definite band matrix A.

This routine is designed to be used in conjunction with SSBGST.

The factorization has the form A = S**T*S where S is a band matrix of the same bandwidth as A and the following structure:

```
S = ( U    )

    ( M  L )
```

where U is upper triangular of order m = (n+kd)/2, and L is lower triangular of order n-m.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the factor S from the split Cholesky factorization A = S**T*S. See Further Details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD+1.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the factorization could not be completed,
because the updated element a(i,i) was negative; the
matrix A is not positive definite.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 7, KD = 2:

S = (s11 s12 s13)

```
    (       s22  s23  s24                )

    (            s33  s34                )

    (                 s44                )

    (            s53  s54  s55           )

    (                 s64  s65  s66      )

    (                      s75  s76  s77 )
```

If UPLO = 'U', the array AB holds:

on entry: on exit:

```
 *    *   a13  a24  a35  a46  a57   *    *   s13  s24  s53  s64  s75
 *   a12  a23  a34  a45  a56  a67   *   s12  s23  s34  s54  s65  s76
a11  a22  a33  a44  a55  a66  a77  s11  s22  s33  s44  s55  s66  s77
```

If UPLO = 'L', the array AB holds:

on entry: on exit:

a11 a22 a33 a44 a55 a66 a77 s11 s22 s33 s44 s55 s66 s77 a21 a32 a43 a54 a65 a76 * s12 s23 s34 s54 s65 s76 * a31 a42 a53 a64 a64 * * s13 s24 s53 s64 s75 * *

Array elements marked * are not used by the routine.

NAME

dpbsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE DPBSV( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DPBSV_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PBSV( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE PBSV_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dpbsv(char uplo, int n, int ndiag, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dpbsv_64(char uplo, long n, long ndiag, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dpbsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite band matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T * U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular band matrix, and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

dpbsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DPBSVX( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPBSVX_64( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBSVX( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE PBSVX_64( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dpbsvx(char fact, char uplo, int n, int ndiag, int nrhs, double *a, int lda, double *af, int ldaf, char equed, double *scale, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dpbsvx_64(char fact, char uplo, long n, long ndiag, long nrhs, double *a, long lda, double *af, long ldaf, char equed, double *scale, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dpbsvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite band matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T * U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular band matrix, and L is a lower
triangular band matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

```
 = 'E':  The matrix A will be equilibrated if necessary, then
copied to AF and factored.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right-hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A (see A). If EQUED = 'Y', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'Y':  Equilibration was done, i.e., A has been replaced by
diag(SCALE) * A * diag(SCALE).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B

that makes X(j) an exact solution).

- **WORK (workspace)**
  dimension(3*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11  a12  a13

     a22  a23  a24

          a33  a34  a35

               a44  a45  a46

                    a55  a56

(aij =conjg(aji))         a66
```

Band storage of the upper triangle of A:

```
 *    *   a13  a24  a35  a46

 *   a12  a23  a34  a45  a56

a11  a22  a33  a44  a55  a66
```

Similarly, if UPLO = 'L' the format of A is as follows:

```
a11  a22  a33  a44  a55  a66

a21  a32  a43  a54  a65   *
```

```
   a31  a42  a53  a64   *    *
```

Array elements marked * are not used by the routine.

NAME

dpbtf2 - compute the Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE DPBTF2( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER N, KD, LDAB, INFO
DOUBLE PRECISION AB(LDAB,*)

SUBROUTINE DPBTF2_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, KD, LDAB, INFO
DOUBLE PRECISION AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE PBTF2( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, KD, LDAB, INFO
REAL(8), DIMENSION(:,:) :: AB

SUBROUTINE PBTF2_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, KD, LDAB, INFO
REAL(8), DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void dpbtf2(char uplo, int n, int kd, double *ab, int ldab, int *info);

void dpbtf2_64(char uplo, long n, long kd, double *ab, long ldab, long *info);

PURPOSE

dpbtf2 computes the Cholesky factorization of a real symmetric positive definite band matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix, U' is the transpose of U, and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **KD (input)**
  The number of super-diagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. KD >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the band matrix A, in the same storage format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, KD = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

dpbtrf - compute the Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE DPBTRF( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DPBTRF_64( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE PBTRF( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE PBTRF_64( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpbtrf(char uplo, int n, int ndiag, double *a, int lda, int *info);

void dpbtrf_64(char uplo, long n, long ndiag, double *a, long lda, long *info);

PURPOSE

dpbtrf computes the Cholesky factorization of a real symmetric positive definite band matrix A.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

Contributed by

Peter Mayes and Giuseppe Radicati, IBM ECSEC, Rome, March 23, 1989

NAME

dpbtrs - solve a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

SYNOPSIS

```
SUBROUTINE DPBTRS( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DPBTRS_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PBTRS( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE PBTRS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dpbtrs(char uplo, int n, int ndiag, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dpbtrs_64(char uplo, long n, long ndiag, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dpbtrs solves a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangular factor stored in A;

  = 'L':  Lower triangular factor stored in A.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dpocon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
  SUBROUTINE DPOCON( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
  CHARACTER * 1 UPLO
  INTEGER N, LDA, INFO
  INTEGER WORK2(*)
  DOUBLE PRECISION ANORM, RCOND
  DOUBLE PRECISION A(LDA,*), WORK(*)

  SUBROUTINE DPOCON_64( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2,
 *      INFO)
  CHARACTER * 1 UPLO
  INTEGER*8 N, LDA, INFO
  INTEGER*8 WORK2(*)
  DOUBLE PRECISION ANORM, RCOND
  DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE POCON( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
 *       [INFO])
  CHARACTER(LEN=1) :: UPLO
  INTEGER :: N, LDA, INFO
  INTEGER, DIMENSION(:) :: WORK2
  REAL(8) :: ANORM, RCOND
  REAL(8), DIMENSION(:) :: WORK
  REAL(8), DIMENSION(:,:) :: A

  SUBROUTINE POCON_64( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK],
 *       [WORK2], [INFO])
  CHARACTER(LEN=1) :: UPLO
  INTEGER(8) :: N, LDA, INFO
  INTEGER(8), DIMENSION(:) :: WORK2
  REAL(8) :: ANORM, RCOND
```

```
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpocon(char uplo, int n, double *a, int lda, double anorm, double *rcond, int *info);

void dpocon_64(char uplo, long n, double *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

dpocon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dpoequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE DPOEQU( N, A, LDA, SCALE, SCOND, AMAX, INFO)
INTEGER N, LDA, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION A(LDA,*), SCALE(*)

SUBROUTINE DPOEQU_64( N, A, LDA, SCALE, SCOND, AMAX, INFO)
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE POEQU( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
INTEGER :: N, LDA, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE POEQU_64( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
INTEGER(8) :: N, LDA, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpoequ(int n, double *a, int lda, double *scale, double *scond, double *amax, int *info);

void dpoequ_64(long n, double *a, long lda, double *scale, double *scond, double *amax, long *info);

PURPOSE

dpoequ computes row and column scalings intended to equilibrate a symmetric positive definite matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The N-by-N symmetric positive definite matrix whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the i-th diagonal element is nonpositive.
```

NAME

dporfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite,

---

SYNOPSIS

```
 SUBROUTINE DPORFS( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPORFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PORFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE PORFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B,
*        [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dporfs(char uplo, int n, int nrhs, double *a, int lda, double *af, int ldaf, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dporfs_64(char uplo, long n, long nrhs, double *a, long lda, double *af, long ldaf, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dporfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPOTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If

XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dposv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE DPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DPOSV_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE POSV( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE POSV_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dposv(char uplo, int n, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dposv_64(char uplo, long n, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dposv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T* U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i of A is not
  positive definite, so the factorization could not be
  completed, and the solution has not been computed.
  ```

NAME

dposvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DPOSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*      SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPOSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*      SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE POSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE POSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dposvx(char fact, char uplo, int n, int nrhs, double *a, int lda, double *af, int ldaf, char equed, double *scale, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dposvx_64(char fact, char uplo, long n, long nrhs, double *a, long lda, double *af, long ldaf, char equed, double *scale, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dposvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T* U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

```
 = 'E':  The matrix A will be equilibrated if necessary, then
copied to AF and factored.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix diag(SCALE)*A*diag(SCALE).

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'Y':  Equilibration was done, i.e., A has been replaced by
diag(SCALE) * A * diag(SCALE).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).

- **WORK (workspace)**
  dimension(3*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, and i is

  < = N: the leading minor of order i of A is
  not positive definite, so the factorization
  could not be completed, and the solution has not
  been computed. RCOND = 0 is returned.
  = N+1: U is nonsingular, but RCOND is less than machine
  precision, meaning that the matrix is singular
  to working precision. Nevertheless, the
  solution and error bounds are computed because
  there are a number of situations where the
  computed solution can be more accurate than the
  value of RCOND would suggest.

NAME

dpotf2 - compute the Cholesky factorization of a real symmetric positive definite matrix A

SYNOPSIS

```
SUBROUTINE DPOTF2( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DPOTF2_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTF2( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE POTF2_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpotf2(char uplo, int n, double *a, int lda, int *info);

void dpotf2_64(char uplo, long n, double *a, long lda, long *info);

PURPOSE

dpotf2 computes the Cholesky factorization of a real symmetric positive definite matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U'*U or A = L*L'.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

NAME

dpotrf - compute the Cholesky factorization of a real symmetric positive definite matrix A

SYNOPSIS

```
SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DPOTRF_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTRF( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE POTRF_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpotrf(char uplo, int n, double *a, int lda, int *info);

void dpotrf_64(char uplo, long n, double *a, long lda, long *info);

PURPOSE

dpotrf computes the Cholesky factorization of a real symmetric positive definite matrix A.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the block version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the factorization could not be
completed.
  ```

NAME

dpotri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
SUBROUTINE DPOTRI( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DPOTRI_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTRI( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE POTRI_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpotri(char uplo, int n, double *a, int lda, int *info);

void dpotri_64(char uplo, long n, double *a, long lda, long *info);

PURPOSE

dpotri computes the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF. On exit, the upper or lower triangle of the (symmetric) inverse of A, overwriting the input factor U or L.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

dpotrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
SUBROUTINE DPOTRS( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DPOTRS_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE POTRS( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE POTRS_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dpotrs(char uplo, int n, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dpotrs_64(char uplo, long n, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dpotrs solves a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dppcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE DPPCON( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER WORK2(*)
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION A(*), WORK(*)

SUBROUTINE DPPCON_64( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 WORK2(*)
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION A(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PPCON( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: WORK2
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: A, WORK

SUBROUTINE PPCON_64( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: WORK2
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void dppcon(char uplo, int n, double *a, double anorm, double *rcond, int *info);

void dppcon_64(char uplo, long n, double *a, double anorm, double *rcond, long *info);

PURPOSE

dppcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dppequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE DPPEQU( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION A(*), SCALE(*)

SUBROUTINE DPPEQU_64( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION A(*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE PPEQU( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: A, SCALE

SUBROUTINE PPEQU_64( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: A, SCALE
```

C INTERFACE

#include <sunperf.h>

void dppequ(char uplo, int n, double *a, double *scale, double *scond, double *amax, int *info);

void dppequ_64(char uplo, long n, double *a, double *scale, double *scond, double *amax, long *info);

PURPOSE

dppequ computes row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). S contains the scale factors, S(i)=1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)=S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the i-th diagonal element is nonpositive.
  ```

NAME

dpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DPPRFS( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPPRFS_64( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PPRFS( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX], FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE PPRFS_64( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX],
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dpprfs(char uplo, int n, int nrhs, double *a, double *af, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dpprfs_64(char uplo, long n, long nrhs, double *a, double *af, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dpprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPPTRF/CPPTRF, packed columnwise in a linear array in the same format as A (see A).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dppsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE DPPSV( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)

SUBROUTINE DPPSV_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PPSV( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE PPSV_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dppsv(char uplo, int n, int nrhs, double *a, double *b, int ldb, int *info);

void dppsv_64(char uplo, long n, long nrhs, double *a, double *b, long ldb, long *info);

PURPOSE

dppsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T* U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

dppsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DPPSVX( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(*), AF(*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPPSVX_64( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(*), AF(*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PPSVX( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: A, AF, SCALE, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE PPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE,
*       B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
```

```
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: A, AF, SCALE, FERR, BERR, WORK
REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dppsvx(char fact, char uplo, int n, int nrhs, double *a, double *af, char equed, double *scale, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dppsvx_64(char fact, char uplo, long n, long nrhs, double *a, double *af, char equed, double *scale, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dppsvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T* U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **AF (input/output)**
  (N*(N+1)/2) If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L', in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED

= 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.

- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, and i is

 <= N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = conjg(aji))

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

dpptrf - compute the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE DPPTRF( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
DOUBLE PRECISION A(*)

SUBROUTINE DPPTRF_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
DOUBLE PRECISION A(*)
```

F95 INTERFACE

```
SUBROUTINE PPTRF( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: A

SUBROUTINE PPTRF_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpptrf(char uplo, int n, double *a, int *info);

void dpptrf_64(char uplo, long n, double *a, long *info);

PURPOSE

dpptrf computes the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

dpptri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE DPPTRI( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
DOUBLE PRECISION A(*)

SUBROUTINE DPPTRI_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
DOUBLE PRECISION A(*)
```

F95 INTERFACE

```
SUBROUTINE PPTRI( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: A

SUBROUTINE PPTRI_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void dpptri(char uplo, int n, double *a, int *info);

void dpptri_64(char uplo, long n, double *a, long *info);

PURPOSE

dpptri computes the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangular factor is stored in A;

 = 'L':  Lower triangular factor is stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise as a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.

  On exit, the upper or lower triangle of the (symmetric) inverse of A, overwriting the input factor U or L.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

dpptrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE DPPTRS( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)

SUBROUTINE DPPTRS_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PPTRS( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE PPTRS_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dpptrs(char uplo, int n, int nrhs, double *a, double *b, int ldb, int *info);

void dpptrs_64(char uplo, long n, long nrhs, double *a, double *b, long ldb, long *info);

PURPOSE

dpptrs solves a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dptcon - compute the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF

SYNOPSIS

```
SUBROUTINE DPTCON( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
INTEGER N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION DIAG(*), OFFD(*), WORK(*)

SUBROUTINE DPTCON_64( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
INTEGER*8 N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION DIAG(*), OFFD(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTCON( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
INTEGER :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: DIAG, OFFD, WORK

SUBROUTINE PTCON_64( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
INTEGER(8) :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: DIAG, OFFD, WORK
```

C INTERFACE

#include <sunperf.h>

void dptcon(int n, double *diag, double *offd, double anorm, double *rcond, int *info);

void dptcon_64(long n, double *diag, double *offd, double anorm, double *rcond, long *info);

PURPOSE

dptcon computes the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF.

`Norm(inv(A))` is computed by a direct method, and the reciprocal of the condition number is computed as

```
RCOND = 1 / (ANORM * norm(inv(A))).
```

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization of A, as computed by SPTTRF.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization of A, as computed by SPTTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The method used is described in Nicholas J. Higham, ``Efficient Algorithms for Computing the Condition Number of a Tridiagonal Matrix'', SIAM J. Sci. Stat. Comput., Vol. 7, No. 1, January 1986.

NAME

dpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor

---

SYNOPSIS

```
SUBROUTINE DPTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)

SUBROUTINE DPTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
REAL(8), DIMENSION(:,:) :: Z

SUBROUTINE PTEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dpteqr(char compz, int n, double *d, double *e, double *z, int ldz, int *info);

void dpteqr_64(char compz, long n, double *d, double *e, double *z, long ldz, long *info);

PURPOSE

dpteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor.

This routine computes the eigenvalues of the positive definite tridiagonal matrix to high relative accuracy. This means that if the eigenvalues range over many orders of magnitude in size, then the small eigenvalues and corresponding eigenvectors will be computed more accurately than, for example, with the standard QR method.

The eigenvectors of a full or band symmetric positive definite matrix can also be found if SSYTRD, SSPTRD, or SSBTRD has been used to reduce this matrix to tridiagonal form. (The reduction to tridiagonal form, however, may preclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix, if these eigenvalues range over many orders of magnitude.)

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvectors of original symmetric
matrix also.  Array Z contains the orthogonal
matrix used to reduce the original matrix to
tridiagonal form.
 = 'I':  Compute eigenvectors of tridiagonal matrix also.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On normal exit, D contains the eigenvalues, in descending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', the orthogonal matrix used in the reduction to tridiagonal form. On exit, if COMPZ = 'V', the orthonormal eigenvectors of the original symmetric matrix; if COMPZ = 'I', the orthonormal eigenvectors of the tridiagonal matrix. If INFO > 0 on exit, Z contains the eigenvectors associated with only the stored eigenvalues. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if COMPZ = 'V' or 'I', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(4*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

 > 0: if INFO = i, and i is:
 < = N the Cholesky factorization of the matrix could
not be performed because the i-th principal minor
was not positive definite.
 > N   the SVD algorithm failed to converge;
if INFO = N+i, i off-diagonal elements of the
bidiagonal factor did not converge to zero.

NAME

dptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DPTRFS( N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB, X,
*      LDX, FERR, BERR, WORK, INFO)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION DIAG(*), OFFD(*), DIAGF(*), OFFDF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPTRFS_64( N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB, X,
*      LDX, FERR, BERR, WORK, INFO)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION DIAG(*), OFFD(*), DIAGF(*), OFFDF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PTRFS( [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B, [LDB],
*       X, [LDX], FERR, BERR, [WORK], [INFO])
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: DIAG, OFFD, DIAGF, OFFDF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE PTRFS_64( [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B, [LDB],
*       X, [LDX], FERR, BERR, [WORK], [INFO])
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: DIAG, OFFD, DIAGF, OFFDF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dptrfs(int n, int nrhs, double *diag, double *offd, double *diagf, double *offdf, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dptrfs_64(long n, long nrhs, double *diag, double *offd, double *diagf, double *offdf, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dptrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.
- **OFFD (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization computed by SPTTRF.
- **OFFDF (input)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization computed by SPTTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dptsv - compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

SYNOPSIS

```
SUBROUTINE DPTSV( N, NRHS, DIAG, SUB, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*), SUB(*), B(LDB,*)

SUBROUTINE DPTSV_64( N, NRHS, DIAG, SUB, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*), SUB(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PTSV( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG, SUB
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE PTSV_64( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG, SUB
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dptsv(int n, int nrhs, double *diag, double *sub, double *b, int ldb, int *info);

void dptsv_64(long n, long nrhs, double *diag, double *sub, double *b, long ldb, long *info);

PURPOSE

dptsv computes the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

A is factored as A = L*D*L**T, and the factored form of A is then used to solve the system of equations.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the factorization A = L*DIAG*L**T.
- **SUB (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A. (SUB can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U**T*DIAG*U factorization of A.)
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the solution has not been
computed.  The factorization has not been completed
unless i  = N.
```

NAME

dptsvx - use the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE DPTSVX( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, INFO)
 CHARACTER * 1 FACT
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION DIAG(*), SUB(*), DIAGF(*), SUBF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DPTSVX_64( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, INFO)
 CHARACTER * 1 FACT
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION DIAG(*), SUB(*), DIAGF(*), SUBF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PTSVX( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [INFO])
 CHARACTER(LEN=1) :: FACT
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: DIAG, SUB, DIAGF, SUBF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE PTSVX_64( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [INFO])
 CHARACTER(LEN=1) :: FACT
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: DIAG, SUB, DIAGF, SUBF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dptsvx(char fact, int n, int nrhs, double *diag, double *sub, double *diagf, double *subf, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dptsvx_64(char fact, long n, long nrhs, double *diag, double *sub, double *diagf, double *subf, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dptsvx uses the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the matrix A is factored as A = L*D*L**T, where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form

```
A = U**T*D*U.
```

2. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, DIAGF and SUBF contain the factored form of A. DIAG, SUB, DIAGF, and SUBF will not be modified. = 'N': The matrix A will be copied to DIAGF and SUBF and factored.
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.
- **SUB (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**T factorization of A. If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**T factorization of A.
- **SUBF (input/output)**
  If FACT = 'F', then SUBF is an input argument and on entry contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A. If FACT = 'N', then SUBF is an output argument and on exit contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 of INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(2*N)
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

dpttrf - compute the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE DPTTRF( N, DIAG, OFFD, INFO)
INTEGER N, INFO
DOUBLE PRECISION DIAG(*), OFFD(*)

SUBROUTINE DPTTRF_64( N, DIAG, OFFD, INFO)
INTEGER*8 N, INFO
DOUBLE PRECISION DIAG(*), OFFD(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRF( [N], DIAG, OFFD, [INFO])
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD

SUBROUTINE PTTRF_64( [N], DIAG, OFFD, [INFO])
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD
```

C INTERFACE

#include <sunperf.h>

void dpttrf(int n, double *diag, double *offd, int *info);

void dpttrf_64(long n, double *diag, double *offd, long *info);

PURPOSE

dpttrf computes the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A. The factorization may also be regarded as having the form A = U'*D*U.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U'*DIAG*U factorization of A.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, the leading minor of order k is not
positive definite; if k  < N, the factorization could not
be completed, while if k  = N, the factorization was
completed, but DIAG(N)  = 0.
```

NAME

dpttrs - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

SYNOPSIS

```
SUBROUTINE DPTTRS( N, NRHS, DIAG, OFFD, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*), OFFD(*), B(LDB,*)

SUBROUTINE DPTTRS_64( N, NRHS, DIAG, OFFD, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*), OFFD(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PTTRS( [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE PTTRS_64( [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dpttrs(int n, int nrhs, double *diag, double *offd, double *b, int ldb, int *info);

void dpttrs_64(long n, long nrhs, double *diag, double *offd, double *b, long ldb, long *info);

PURPOSE

dpttrs solves a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF. D is a diagonal matrix specified in the vector D, L is a unit bidiagonal matrix whose subdiagonal is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the factorization A = U'*DIAG*U.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -k, the k-th argument had an illegal value
```

NAME

dptts2 - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

SYNOPSIS

```
SUBROUTINE DPTTS2( N, NRHS, D, E, B, LDB)
INTEGER N, NRHS, LDB
DOUBLE PRECISION D(*), E(*), B(LDB,*)

SUBROUTINE DPTTS2_64( N, NRHS, D, E, B, LDB)
INTEGER*8 N, NRHS, LDB
DOUBLE PRECISION D(*), E(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE DPTTS2( N, NRHS, D, E, B, LDB)
INTEGER :: N, NRHS, LDB
REAL(8), DIMENSION(:) :: D, E
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE DPTTS2_64( N, NRHS, D, E, B, LDB)
INTEGER(8) :: N, NRHS, LDB
REAL(8), DIMENSION(:) :: D, E
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dptts2(int n, int nrhs, double *d, double *e, double *b, int ldb);

void dptts2_64(long n, long nrhs, double *d, double *e, double *b, long ldb);

PURPOSE

dptts2 solves a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF. D is a diagonal matrix specified in the vector D, L is a unit bidiagonal matrix whose subdiagonal is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **D (input)**
  The n diagonal elements of the diagonal matrix D from the L*D*L' factorization of A.
- **E (input)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*D*L' factorization of A. E can also be regarded as the superdiagonal of the unit bidiagonal factor U from the factorization A = U'*D*U.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

NAME

dqdota - compute a double precision constant plus an extended precision constant plus the extended precision dot product of two double precision vectors x and y.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DQDOTA( N, DB, QC, DX, INCX, DY, INCY)
INTEGER N, INCX, INCY
REAL * 16 QC
DOUBLE PRECISION DB
DOUBLE PRECISION DX(*), DY(*)

DOUBLE PRECISION FUNCTION DQDOTA_64( N, DB, QC, DX, INCX, DY, INCY)
INTEGER*8 N, INCX, INCY
REAL * 16 QC
DOUBLE PRECISION DB
DOUBLE PRECISION DX(*), DY(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION DQDOTA( N, DB, QC, DX, INCX, DY, INCY)
INTEGER :: N, INCX, INCY
REAL(16) :: QC
REAL(8) :: DB
REAL(8), DIMENSION(:) :: DX, DY

REAL(8) FUNCTION DQDOTA_64( N, DB, QC, DX, INCX, DY, INCY)
INTEGER(8) :: N, INCX, INCY
REAL(16) :: QC
REAL(8) :: DB
REAL(8), DIMENSION(:) :: DX, DY
```

C INTERFACE

#include <sunperf.h>

double dqdota(int n, double db, long double *qc, double *dx, int incx, double *dy, int incy);

double dqdota_64(long n, double db, long double *qc, double *dx, long incx, double *dy, long incy);

PURPOSE

dqdota compute a double precision constant plus an extended precision constant plus the extended precision dot product of two double precision vectors x and y.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N < = 0 then the function returns the value DB+QC. Unchanged on exit.
- **DB (input)**
  On entry, the constant that is added to the dot product before the result is returned. Unchanged on exit.
- **QC (input/output)**
  On entry, the extended precision constant to be added to the dot product. On exit, the extended precision result.
- **DX (input)**
  On entry, the incremented array DX must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of DX. INCX must not be zero. Unchanged on exit.
- **DY (input)**
  On entry, the incremented array DY must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of DY. INCY must not be zero. Unchanged on exit.

NAME

dqdoti - compute a constant plus the extended precision dot product of two double precision vectors x and y.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DQDOTI( N, DB, QC, DX, INCX, DY, INCY)
INTEGER N, INCX, INCY
REAL * 16 QC
DOUBLE PRECISION DB
DOUBLE PRECISION DX(*), DY(*)

DOUBLE PRECISION FUNCTION DQDOTI_64( N, DB, QC, DX, INCX, DY, INCY)
INTEGER*8 N, INCX, INCY
REAL * 16 QC
DOUBLE PRECISION DB
DOUBLE PRECISION DX(*), DY(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION DQDOTI( N, DB, QC, DX, INCX, DY, INCY)
INTEGER :: N, INCX, INCY
REAL(16) :: QC
REAL(8) :: DB
REAL(8), DIMENSION(:) :: DX, DY

REAL(8) FUNCTION DQDOTI_64( N, DB, QC, DX, INCX, DY, INCY)
INTEGER(8) :: N, INCX, INCY
REAL(16) :: QC
REAL(8) :: DB
REAL(8), DIMENSION(:) :: DX, DY
```

C INTERFACE

#include <sunperf.h>

double dqdoti(int n, double db, long double *qc, double *dx, int incx, double *dy, int incy);

double dqdoti_64(long n, double db, long double *qc, double *dx, long incx, double *dy, long incy);

PURPOSE

dqdoti computes a constant plus the double precision dot product of x and y where x and y are double precision n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N < = 0 then the function returns the value DB. Unchanged on exit.
- **DB (input)**
  On entry, the constant that is added to the dot product before the result is returned. Unchanged on exit.
- **QC (input/output)**
  On exit, the extended precision result.
- **DX (input)**
  On entry, the incremented array DX must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of DX. INCX must not be zero. Unchanged on exit.
- **DY (input)**
  On entry, the incremented array DY must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of DY. INCY must not be zero. Unchanged on exit.

NAME

drot - Apply a Given's rotation constructed by SROTG.

SYNOPSIS

```
SUBROUTINE DROT( N, X, INCX, Y, INCY, C, S)
INTEGER N, INCX, INCY
DOUBLE PRECISION C, S
DOUBLE PRECISION X(*), Y(*)

SUBROUTINE DROT_64( N, X, INCX, Y, INCY, C, S)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION C, S
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], X, [INCX], Y, [INCY], C, S)
INTEGER :: N, INCX, INCY
REAL(8) :: C, S
REAL(8), DIMENSION(:) :: X, Y

SUBROUTINE ROT_64( [N], X, [INCX], Y, [INCY], C, S)
INTEGER(8) :: N, INCX, INCY
REAL(8) :: C, S
REAL(8), DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void drot(int n, double *x, int incx, double *y, int incy, double c, double s);

void drot_64(long n, double *x, long incx, double *y, long incy, double c, double s);

PURPOSE

drot Apply a Given's rotation constructed by SROTG.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **C (input)**
  On entry, the C rotation value constructed by SROTG. Unchanged on exit.
- **S (input)**
  On entry, the S rotation value constructed by SROTG. Unchanged on exit.

NAME

drotg - Construct a Given's plane rotation

SYNOPSIS

```
SUBROUTINE DROTG( A, B, C, S)
DOUBLE PRECISION A, B, C, S

SUBROUTINE DROTG_64( A, B, C, S)
DOUBLE PRECISION A, B, C, S
```

F95 INTERFACE

```
SUBROUTINE ROTG( A, B, C, S)
REAL(8) :: A, B, C, S

SUBROUTINE ROTG_64( A, B, C, S)
REAL(8) :: A, B, C, S
```

C INTERFACE

#include <sunperf.h>

void drotg(double *a, double *b, double *c, double *s);

void drotg_64(double *a, double *b, double *c, double *s);

PURPOSE

drotg Construct a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **A (input/output)**
  On entry, A contains the entry in the first vector that corresponds to the element to be annihilated in the second vector. On exit, contains the nonzero element of the rotated vector.
- **B (input/output)**
  On entry, B contains the entry to be annihilated in the second vector. On exit, contains either S or 1/C depending on which of the input values of A and B is larger.
- **C (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.
- **S (output)**
  See the description of C.

NAME

droti - Apply an indexed Givens rotation.

SYNOPSIS

```
SUBROUTINE DROTI(NZ, X, INDX, Y, C, S)

INTEGER NZ
INTEGER INDX(*)
DOUBLE PRECISION C, S
DOUBLE PRECISION X(*), Y(*)

SUBROUTINE DROTI_64(NZ, X, INDX, Y, C, S)

INTEGER*8 NZ
INTEGER*8 INDX(*)
DOUBLE PRECISION C, S
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE SUBROUTINE ROTI([NZ], X, INDX, Y, C, S)

```
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX
REAL(8) :: C, S
REAL(8), DIMENSION(:) :: X, Y

SUBROUTINE ROTI_64([NZ], X, INDX, Y, C, S)

INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
REAL(8) :: C, S
REAL(8), DIMENSION(:) :: X, Y
```

PURPOSE

DROTI - Applies a Givens rotation to a sparse vector x stored in compressed form and another vector y in full storage form

```
do i = 1, n
  temp = -s * x(i) + c * y(indx(i))
  x(i) = c * x(i) + s * y(indx(i))
  y(indx(i)) = temp
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values of the compressed form.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input/output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

C (input)

Scalar defining the Givens rotation

S (input)

Scalar defining the Givens rotation

NAME

drotm - Apply a Gentleman's modified Given's rotation constructed by SROTMG.

SYNOPSIS

```
SUBROUTINE DROTM( N, X, INCX, Y, INCY, PARAM)
INTEGER N, INCX, INCY
DOUBLE PRECISION X(*), Y(*), PARAM(*)

SUBROUTINE DROTM_64( N, X, INCX, Y, INCY, PARAM)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION X(*), Y(*), PARAM(*)
```

F95 INTERFACE

```
SUBROUTINE ROTM( [N], X, [INCX], Y, [INCY], PARAM)
INTEGER :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y, PARAM

SUBROUTINE ROTM_64( [N], X, [INCX], Y, [INCY], PARAM)
INTEGER(8) :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y, PARAM
```

C INTERFACE

#include <sunperf.h>

void drotm(int n, double *x, int incx, double *y, int incy, double *param);

void drotm_64(long n, double *x, long incx, double *y, long incy, double *param);

PURPOSE

drotm Apply a Given's rotation constructed by SROTMG.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, X is overwritten by the updated vector x.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **PARAM (input)**
  On entry, the rotation values constructed by SROTMG. Unchanged on exit.

NAME

drotmg - Construct a Gentleman's modified Given's plane rotation

---

SYNOPSIS

```
SUBROUTINE DROTMG( D1, D2, B1, B2, PARAM)
DOUBLE PRECISION D1, D2, B1, B2
DOUBLE PRECISION PARAM(*)

SUBROUTINE DROTMG_64( D1, D2, B1, B2, PARAM)
DOUBLE PRECISION D1, D2, B1, B2
DOUBLE PRECISION PARAM(*)
```

F95 INTERFACE

```
SUBROUTINE ROTMG( D1, D2, B1, B2, PARAM)
REAL(8) :: D1, D2, B1, B2
REAL(8), DIMENSION(:) :: PARAM

SUBROUTINE ROTMG_64( D1, D2, B1, B2, PARAM)
REAL(8) :: D1, D2, B1, B2
REAL(8), DIMENSION(:) :: PARAM
```

C INTERFACE

#include <sunperf.h>

void drotmg(double *d1, double *d2, double *b1, double *b2, double *param);

void drotmg_64(double *d1, double *d2, double *b1, double *b2, double *param);

PURPOSE

drotmg Construct Gentleman's modified a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **D1 (input/output)**
  On entry, the first diagonal entry in the H matrix. On exit, changed to reflect the effect of the transformation.
- **D2 (input/output)**
  On entry, the second diagonal entry in the H matrix. On exit, changed to reflect the effect of the transformation.
- **B1 (input/output)**
  On entry, the first element of the vector to which the H matrix is applied. On exit, changed to reflect the effect of the transformation.
- **B2 (input/output)**
  On entry, the second element of the vector to which the H matrix is applied. Unchanged on exit.
- **PARAM (input/output)**
  On exit, `PARAM(1)` describes the form of the rotation matrix H, and `PARAM(2..5)` contain the H matrix.

  If `PARAM(1)` = -2 then H = I and no elements of PARAM are modified.

  If `PARAM(1)` = -1 then `PARAM(2)` = h11, `PARAM(3)` = h21, `PARAM(4)` = h12, and `PARAM(5)` = h22.

  If `PARAM(1)` = 0 then h11 = h22 = 1, `PARAM(3)` = h21, and `PARAM(4)` = h12.

  If `PARAM(1)` = 1 then h12 = 1, h21 = -1, `PARAM(2)` = h11, and `PARAM(5)` = h22.

NAME

dsbev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE DSBEV( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, NDIAG, LDA, LDZ, INFO
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBEV_64( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, NDIAG, LDA, LDZ, INFO
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEV( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, NDIAG, LDA, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, Z

 SUBROUTINE SBEV_64( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, NDIAG, LDA, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void dsbev(char jobz, char uplo, int n, int ndiag, double *a, int lda, double *w, double *z, int ldz, int *info);

void dsbev_64(char jobz, char uplo, long n, long ndiag, double *a, long lda, double *w, double *z, long ldz, long *info);

PURPOSE

dsbev computes all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG + 1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  dimension(MAX(1,3*N-2))
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, the algorithm failed to converge; i
  off-diagonal elements of an intermediate tridiagonal
  form did not converge to zero.

NAME

dsbevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE DSBEVD( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION AB(LDAB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBEVD_64( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION AB(LDAB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEVD( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, Z

 SUBROUTINE SBEVD_64( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, Z
```

C INTERFACE

#include <sunperf.h>

void dsbevd(char jobz, char uplo, int n, int kd, double *ab, int ldab, double *w, double *z, int ldz, double *work, int lwork, int *info);

void dsbevd_64(char jobz, char uplo, long n, long kd, double *ab, long ldab, double *w, double *z, long ldz, double *work, long lwork, long *info);

PURPOSE

dsbevd computes all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, AB is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows KD and KD+1 of AB, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of AB.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD + 1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK must be at least 1. If JOBZ = 'N' and N > 2, LWORK must be at least 2*N. If JOBZ = 'V' and N > 2, LWORK must be at least (1 + 5*N + 2*N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array LIWORK. If JOBZ = 'N' or N <= 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 2, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

dsbevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE DSBEVX( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ, VL,
*      VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(LDA,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBEVX_64( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ,
*      VL, VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(LDA,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEVX( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q, [LDQ],
*       VL, VU, IL, IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, Q, Z

 SUBROUTINE SBEVX_64( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q,
*       [LDQ], VL, VU, IL, IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK],
*       [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
```

```
INTEGER(8) :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
REAL(8) :: VL, VU, ABTOL
REAL(8), DIMENSION(:) :: W, WORK
REAL(8), DIMENSION(:,:) :: A, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dsbevx(char jobz, char range, char uplo, int n, int ndiag, double *a, int lda, double *q, int ldq, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, double *z, int ldz, int *ifail, int *info);

void dsbevx_64(char jobz, char range, char uplo, long n, long ndiag, double *a, long lda, double *q, long ldq, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dsbevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the

array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd).

On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG + 1.
- **Q (output)**
  If JOBZ = 'V', the N-by-N orthogonal matrix used in the reduction to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'V', then LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 <= NFOUND <= N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least max(1,NFOUND) columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  dimension(7*N)
- **IWORK2 (workspace)**

- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.

  > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

dsbgst - reduce a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

---

SYNOPSIS

```
 SUBROUTINE DSBGST( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*       LDX, WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDX, INFO
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)

 SUBROUTINE DSBGST_64( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*       LDX, WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDX, INFO
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGST( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        X, [LDX], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, X

 SUBROUTINE SBGST_64( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        X, [LDX], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, X
```

C INTERFACE

#include <sunperf.h>

void dsbgst(char vect, char uplo, int n, int ka, int kb, double *ab, int ldab, double *bb, int ldbb, double *x, int ldx, int *info);

void dsbgst_64(char vect, char uplo, long n, long ka, long kb, double *ab, long ldab, double *bb, long ldbb, double *x, long ldx, long *info);

PURPOSE

dsbgst reduces a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y, such that C has the same bandwidth as A.

B must have been previously factorized as S**T*S by SPBSTF, using a split Cholesky factorization. A is overwritten by C = X**T*A*X, where X = S**(-1)*Q and Q is an orthogonal matrix chosen to preserve the bandwidth of A.

ARGUMENTS

- **VECT (input)**

  = 'N': do not form the transformation matrix X;

  = 'V': form X.

- **UPLO (input)**

  = 'U': Upper triangle of A is stored;

  = 'L': Lower triangle of A is stored.

- **N (input)**
  The order of the matrices A and B. N >= 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= KB >= 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the transformed matrix X**T*A*X, stored in the same format as A.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.

- **BB (input)**
  The banded factor S from the split Cholesky factorization of B, as returned by SPBSTF, stored in the first KB+1

rows of the array.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **X (output)**
  If VECT = 'V', the n-by-n matrix X. If VECT = 'N', the array X is not referenced.
- **LDX (input)**
  The leading dimension of the array X. LDX >= `max(1,N)` if VECT = 'V'; LDX >= 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

dsbgv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE DSBGV( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*       LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, INFO
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBGV_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*       Z, LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, INFO
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGV( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB], W,
*        Z, [LDZ], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, Z

 SUBROUTINE SBGV_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        W, Z, [LDZ], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, Z
```

C INTERFACE

#include <sunperf.h>

void dsbgv(char jobz, char uplo, int n, int ka, int kb, double *ab, int ldab, double *bb, int ldbb, double *w, double *z, int ldz, int *info);

void dsbgv_64(char jobz, char uplo, long n, long ka, long kb, double *ab, long ldab, double *bb, long ldbb, double *w, double *z, long ldz, long *info);

PURPOSE

dsbgv computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` =

`B(i,j)` for `max(1,j-kb)` < =i < =j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j < =i < =min(n,j+kb).

On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then SPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

NAME

dsbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE DSBGVD( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*      LDZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBGVD_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*      Z, LDZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGVD( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, Z

 SUBROUTINE SBGVD_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
```

```
REAL(8), DIMENSION(:,:) :: AB, BB, Z
```

C INTERFACE

#include <sunperf.h>

void dsbgvd(char jobz, char uplo, int n, int ka, int kb, double *ab, int ldab, double *bb, int ldbb, double *w, double *z, int ldz, double *work, int lwork, int *info);

void dsbgvd_64(char jobz, char uplo, long n, long ka, long kb, double *ab, long ldab, double *bb, long ldbb, double *w, double *z, long ldz, double *work, long lwork, long *info);

PURPOSE

dsbgvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(ka+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= 3*N. If JOBZ = 'V' and N > 1, LWORK >= 1 + 5*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if LIWORK > 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 <= N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then SPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

dsbgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE DSBGVX( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB, LDBB,
*      Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSBGVX_64( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB,
*      LDBB, Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGVX( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, Q, Z
```

```
 SUBROUTINE SBGVX_64( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: AB, BB, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dsbgvx(char jobz, char range, char uplo, int n, int ka, int kb, double *ab, int ldab, double *bb, int ldbb, double *q, int ldq, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *ifail, int *info);

void dsbgvx_64(char jobz, char range, char uplo, long n, long ka, long kb, double *ab, long ldab, double *bb, long ldbb, double *q, long ldq, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dsbgvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(ka+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **Q (output)**
  If JOBZ = 'V', the n-by-n matrix used in the reduction of A*x = (lambda)*B*x to standard form, i.e. C*x = (lambda)*x, and consequently C to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'N', LDQ >= 1. If JOBZ = 'V', LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 <= M <= N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**

If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (input)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvalues that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0 : successful exit

 < 0 : if INFO  = -i, the i-th argument had an illegal value

 < = N: if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in IFAIL.
 > N : SPBSTF returned an error code; i.e.,
if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

dsbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE DSBMV( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)

 SUBROUTINE DSBMV_64( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA,
*      Y, INCY)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INCX, INCY
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE SBMV( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX], BETA,
*       Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE SBMV_64( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INCX, INCY
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:) :: X, Y
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsbmv(char uplo, int n, int ndiag, double alpha, double *a, int lda, double *x, int incx, double beta, double *y, int incy);

void dsbmv_64(char uplo, long n, long ndiag, double alpha, double *a, long lda, double *x, long incx, double beta, double *y, long incy);

PURPOSE

dsbmv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with ndiag super-diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the band matrix A is being supplied as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is being supplied.

  UPLO = 'L' or 'l' The lower triangular part of A is being supplied.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry, NDIAG specifies the number of super-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

  Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
   DO 20, J = 1, N
     M = 1 - J
     DO 10, I = J, MIN( N, J + NDIAG )
       A( M + I, J ) = matrix( I, J )
10   CONTINUE
20 CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dsbtrd - reduce a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
 SUBROUTINE DSBTRD( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ, WORK,
*      INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER N, KD, LDAB, LDQ, INFO
 DOUBLE PRECISION AB(LDAB,*), D(*), E(*), Q(LDQ,*), WORK(*)

 SUBROUTINE DSBTRD_64( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ,
*      WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER*8 N, KD, LDAB, LDQ, INFO
 DOUBLE PRECISION AB(LDAB,*), D(*), E(*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBTRD( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER :: N, KD, LDAB, LDQ, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: AB, Q

 SUBROUTINE SBTRD_64( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER(8) :: N, KD, LDAB, LDQ, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: AB, Q
```

C INTERFACE

#include <sunperf.h>

void dsbtrd(char vect, char uplo, int n, int kd, double *ab, int ldab, double *d, double *e, double *q, int ldq, int *info);

void dsbtrd_64(char vect, char uplo, long n, long kd, double *ab, long ldab, double *d, double *e, double *q, long ldq, long *info);

PURPOSE

dsbtrd reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **VECT (input)**

  ```
  = 'N':  do not form Q;

  = 'V':  form Q;

  = 'U':  update a matrix X, by forming X*Q.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD >= 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). On exit, the diagonal elements of AB are overwritten by the diagonal elements of the tridiagonal matrix T; if KD > 0, the elements on the first superdiagonal (if UPLO = 'U') or the first subdiagonal (if UPLO = 'L') are overwritten by the off-diagonal elements of T; the rest of AB is overwritten by values generated during the reduction.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.

- **D (output)**
  The diagonal elements of the tridiagonal matrix T.

- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `T(i,i+1)` if UPLO = 'U'; `E(i)` = `T(i+1,i)` if

UPLO = 'L'.

- **Q (input/output)**
  On entry, if VECT = 'U', then Q must contain an N-by-N matrix X; if VECT = 'N' or 'V', then Q need not be set.

  On exit: if VECT = 'V', Q contains the N-by-N orthogonal matrix Q; if VECT = 'U', Q contains the product X*Q; if VECT = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1, and LDQ >= N if VECT = 'V' or 'U'.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Modified by Linda Kaufman, Bell Labs.

NAME

dscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE DSCAL( N, ALPHA, Y, INCY)
INTEGER N, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION Y(*)

SUBROUTINE DSCAL_64( N, ALPHA, Y, INCY)
INTEGER*8 N, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION Y(*)
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
INTEGER :: N, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: Y

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
INTEGER(8) :: N, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: Y
```

C INTERFACE

#include <sunperf.h>

void dscal(int n, double alpha, double *y, int incy);

void dscal_64(long n, double alpha, double *y, long incy);

PURPOSE

dscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

dsctr - Scatters elements from x into y.

SYNOPSIS

```
SUBROUTINE DSCTR(NZ, X, INDX, Y)

DOUBLE PRECISION X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE DSCTR_64(NZ, X, INDX, Y)

DOUBLE PRECISION X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE SCTR([NZ], X, INDX, Y)

```
REAL(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE SCTR_64([NZ], X, INDX, Y)

REAL(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

DSCTR - Scatters the components of a sparse vector x stored in compressed form into specified components of a vector y in full storage form.

```
do i = 1, n
  y(indx(i)) = x(i)
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values to be scattered from compressed form into full storage form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector whose elements specified by indx have been set to the corresponding entries of x. Only the elements corresponding to the indices in indx have been modified.

NAME

dsdot - compute the double precision dot product of two single precision vectors x and y.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DSDOT( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
REAL X(*), Y(*)

DOUBLE PRECISION FUNCTION DSDOT_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
REAL X(*), Y(*)
```

F95 INTERFACE

```
REAL(8) FUNCTION DSDOT( N, X, INCX, Y, INCY)
INTEGER :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y

REAL(8) FUNCTION DSDOT_64( N, X, INCX, Y, INCY)
INTEGER(8) :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

double dsdot(int n, float *x, int incx, float *y, int incy);

double dsdot_64(long n, float *x, long incx, float *y, long incy);

PURPOSE

dsdot compute the double precision dot product of x and y where x and y are single precision n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

dsecnd - return the user time for a process in seconds =head1 SYNOPSIS

```
DOUBLE PRECISION FUNCTION DSECND( )

DOUBLE PRECISION FUNCTION DSECND_64( )
```

F95 INTERFACE

```
REAL(8) FUNCTION DSECND( )

REAL(8) FUNCTION DSECND_64( )
```

C INTERFACE

#include <sunperf.h>

double dsecnd();

double dsecnd_64();

PURPOSE

dsecnd returns the user time for a process in seconds. This version gets the time from the system function ETIME.

ARGUMENTS

NAME

dsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE DSINQB( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DSINQB_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQB( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE SINQB_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dsinqb(int n, double *x, double *wsave);

void dsinqb_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave sine synthesis of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) for scalar subroutines, initialized by SINQI.

NAME

dsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE DSINQF( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DSINQF_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQF( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE SINQF_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dsinqf(int n, double *x, double *wsave);

void dsinqf_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave sine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) for scalar subroutines, initialized by SINQI.

NAME

dsinqi - initialize the array xWSAVE, which is used in both SINQF and SINQB.

SYNOPSIS

```
SUBROUTINE DSINQI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DSINQI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE SINQI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dsinqi(int n, double *wsave);

void dsinqi_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. SINQI needs to be called only once to initialize WSAVE before calling SINQF and/or SINQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dsint - compute the discrete Fourier sine transform of an odd sequence. The SINT transforms are unnormalized inverses of themselves, so a call of SINT followed by another call of SINT will multiply the input sequence by 2 * (N+1).

SYNOPSIS

```
SUBROUTINE DSINT( N, X, WSAVE)
INTEGER N
DOUBLE PRECISION X(*), WSAVE(*)

SUBROUTINE DSINT_64( N, X, WSAVE)
INTEGER*8 N
DOUBLE PRECISION X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINT( [N], X, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: X, WSAVE

SUBROUTINE SINT_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void dsint(int n, double *x, double *wsave);

void dsint_64(long n, double *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N+1 is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the sine transform of the input.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least int(2.5 * N + 15) initialized by SINTI.

NAME

dsinti - initialize the array WSAVE, which is used in subroutine SINT.

SYNOPSIS

```
SUBROUTINE DSINTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE DSINTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE SINTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void dsinti(int n, double *wsave);

void dsinti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2N + N/2 + 15) or greater. SINTI is called once to initialize WSAVE before calling SINT and need not be called again between calls to SINT if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

dspcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
 SUBROUTINE DSPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, IWORK2,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER N, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(*), WORK(*)

 SUBROUTINE DSPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: A, WORK

 SUBROUTINE SPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
 REAL(8) :: ANORM, RCOND
```

```
REAL(8), DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void dspcon(char uplo, int n, double *a, int *ipivot, double anorm, double *rcond, int *info);

void dspcon_64(char uplo, long n, double *a, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

dspcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **IWORK2 (workspace)**
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dspev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
SUBROUTINE DSPEV( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER N, LDZ, INFO
DOUBLE PRECISION A(*), W(*), Z(LDZ,*), WORK(*)

SUBROUTINE DSPEV_64( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION A(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SPEV( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: A, W, WORK
REAL(8), DIMENSION(:,:) :: Z

SUBROUTINE SPEV_64( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: A, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspev(char jobz, char uplo, int n, double *a, double *w, double *z, int ldz, int *info);

void dspev_64(char jobz, char uplo, long n, double *a, double *w, double *z, long ldz, long *info);

PURPOSE

dspev computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

dspevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE DSPEVD( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION AP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSPEVD_64( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK,
*      IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION AP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPEVD( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: AP, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE SPEVD_64( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: AP, W, WORK
 REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspevd(char jobz, char uplo, int n, double *ap, double *w, double *z, int ldz, double *work, int lwork, int *info);

void dspevd_64(char jobz, char uplo, long n, double *ap, double *w, double *z, long ldz, double *work, long lwork, long *info);

PURPOSE

dspevd computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, AP is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**

The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least 2*N. If JOBZ = 'V' and N > 1, LWORK must be at least 1 + 6*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N < = 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

dspevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE DSPEVX( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU, ABTOL,
*      NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSPEVX_64( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPEVX( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU, ABTOL,
*       NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: A, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE SPEVX_64( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
```

```
REAL(8), DIMENSION(:) :: A, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspevx(char jobz, char range, char uplo, int n, double *a, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, double *z, int ldz, int *ifail, int *info);

void dspevx_64(char jobz, char range, char uplo, long n, double *a, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dspevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. Eigenvalues/vectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  If INFO = 0, the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(8*N)`
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

dspgst - reduce a real symmetric-definite generalized eigenproblem to standard form, using packed storage

---

SYNOPSIS

```
SUBROUTINE DSPGST( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, INFO
DOUBLE PRECISION AP(*), BP(*)

SUBROUTINE DSPGST_64( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, INFO
DOUBLE PRECISION AP(*), BP(*)
```

F95 INTERFACE

```
SUBROUTINE SPGST( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, INFO
REAL(8), DIMENSION(:) :: AP, BP

SUBROUTINE SPGST_64( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, INFO
REAL(8), DIMENSION(:) :: AP, BP
```

C INTERFACE

#include <sunperf.h>

void dspgst(int itype, char uplo, int n, double *ap, double *bp, int *info);

void dspgst_64(long itype, char uplo, long n, double *ap, double *bp, long *info);

PURPOSE

dspgst reduces a real symmetric-definite generalized eigenproblem to standard form, using packed storage.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**T)*A*inv(U)` or `inv(L)*A*inv(L**T)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**T or L**T*A*L.

B must have been previously factorized as U**T*U or L*L**T by SPPTRF.

ARGUMENTS

- **ITYPE (input)**

```
= 1: compute inv(U**T)*A*inv(U) or inv(L)*A*inv(L**T);

= 2 or 3: compute U*A*U**T or L**T*A*L.
```

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored and B is factored as
U**T*U;
 = 'L':  Lower triangle of A is stored and B is factored as
L*L**T.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.
- **BP (input)**
  The triangular factor from the Cholesky factorization of B, stored in the same format as A, as returned by SPPTRF.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dspgv - compute all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSPGV( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDZ, INFO
 DOUBLE PRECISION A(*), B(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSPGV_64( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDZ, INFO
 DOUBLE PRECISION A(*), B(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGV( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDZ, INFO
 REAL(8), DIMENSION(:) :: A, B, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE SPGV_64( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDZ, INFO
 REAL(8), DIMENSION(:) :: A, B, W, WORK
 REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspgv(int itype, char jobz, char uplo, int n, double *a, double *b, double *w, double *z, int ldz, int *info);

void dspgv_64(long itype, char jobz, char uplo, long n, double *a, double *b, double *w, double *z, long ldz, long *info);

PURPOSE

dspgv computes all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  (N*(N+1)/2) On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of A are destroyed.

- **B (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array B as follows: if UPLO = 'U', B(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', B(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEV returned an error code:

 < = N:  if INFO  = i, SSPEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero.
 > N:   if INFO  = n + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

dspgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSPGVD( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSPGVD_64( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGVD( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ], [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: AP, BP, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE SPGVD_64( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ],
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: AP, BP, W, WORK
```

```
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspgvd(int itype, char jobz, char uplo, int n, double *ap, double *bp, double *w, double *z, int ldz, int *info);

void dspgvd_64(long itype, char jobz, char uplo, long n, double *ap, double *bp, double *w, double *z, long ldz, long *info);

PURPOSE

dspgvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th

column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = A(i,j) for 1 <=i <=j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = A(i,j) for j <=i <=n.

On exit, the contents of AP are destroyed.

- **BP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = B(i,j) for 1 <=i <=j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = B(i,j) for j <=i <=n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= 2*N. If JOBZ = 'V' and N > 1, LWORK >= 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEVD returned an error code:

 <= N:  if INFO  = i, SSPEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

dspgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSPGVX( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU, IL,
*      IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSPGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU,
*      IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGVX( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], [WORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: AP, BP, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE SPGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU,
*       IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
```

```
INTEGER(8) :: ITYPE, N, IL, IU, M, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
REAL(8) :: VL, VU, ABSTOL
REAL(8), DIMENSION(:) :: AP, BP, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dspgvx(int itype, char jobz, char range, char uplo, int n, double *ap, double *bp, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *ifail, int *info);

void dspgvx_64(long itype, char jobz, char range, char uplo, long n, double *ap, double *bp, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dspgvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed storage, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

= 'U': Upper triangle of A and B are stored;

= 'L': Lower triangle of A and B are stored.

- **N (input)**
  The order of the matrix pencil (A,B). N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

  If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  `dimension(8*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEVX returned an error code:

 < = N:  if INFO  = i, SSPEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

dspmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE DSPMV( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY
DOUBLE PRECISION ALPHA, BETA
DOUBLE PRECISION A(*), X(*), Y(*)

SUBROUTINE DSPMV_64( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION ALPHA, BETA
DOUBLE PRECISION A(*), X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SPMV( UPLO, N, ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY
REAL(8) :: ALPHA, BETA
REAL(8), DIMENSION(:) :: A, X, Y

SUBROUTINE SPMV_64( UPLO, N, ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY
REAL(8) :: ALPHA, BETA
REAL(8), DIMENSION(:) :: A, X, Y
```

C INTERFACE

#include <sunperf.h>

void dspmv(char uplo, int n, double alpha, double *a, double *x, int incx, double beta, double *y, int incy);

void dspmv_64(char uplo, long n, double alpha, double *a, double *x, long incx, double beta, double *y, long incy);

PURPOSE

dspmv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dspr - perform the symmetric rank 1 operation A := alpha*x*x' + A

SYNOPSIS

```
SUBROUTINE DSPR( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
INTEGER N, INCX
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), A(*)

SUBROUTINE DSPR_64( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), A(*)
```

F95 INTERFACE

```
SUBROUTINE SPR( UPLO, N, ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, A

SUBROUTINE SPR_64( UPLO, N, ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, A
```

C INTERFACE

#include <sunperf.h>

void dspr(char uplo, int n, double alpha, double *x, int incx, double *a);

void dspr_64(char uplo, long n, double alpha, double *x, long incx, double *a);

PURPOSE

dspr performs the symmetric rank 1 operation A := alpha*x*x' + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix.

NAME

dspr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

SYNOPSIS

```
SUBROUTINE DSPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(*)

SUBROUTINE DSPR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(*)
```

F95 INTERFACE

```
SUBROUTINE SPR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y, A

SUBROUTINE SPR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y, A
```

C INTERFACE

#include <sunperf.h>

void dspr2(char uplo, int n, double alpha, double *x, int incx, double *y, int incy, double *a);

void dspr2_64(char uplo, long n, double alpha, double *x, long incx, double *y, long incy, double *a);

PURPOSE

dspr2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix.

NAME

dsprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DSPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DSPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE SPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dsprfs(char uplo, int n, int nrhs, double *a, double *af, int *ipivot, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dsprfs_64(char uplo, long n, long nrhs, double *a, double *af, long *ipivot, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dsprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SSPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If

XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dspsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE DSPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(*), B(LDB,*)

SUBROUTINE DSPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SPSV( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE SPSV_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dspsv(char uplo, int n, int nrhs, double *a, int *ipivot, double *b, int ldb, int *info);

void dspsv_64(char uplo, long n, long nrhs, double *a, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dspsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by SSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = aji)

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

dspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE DSPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DSPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X,
*       [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE SPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
```

```
CHARACTER(LEN=1) :: FACT, UPLO
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: A, AF, FERR, BERR, WORK
REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dspsvx(char fact, char uplo, int n, int nrhs, double *a, double *af, int *ipivot, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dspsvx_64(char fact, char uplo, long n, long nrhs, double *a, double *af, long *ipivot, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dspsvx uses the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**T, if UPLO = 'U', or

```
    A = L * D * L**T,  if UPLO = 'L',

where U (or L) is a product of permutation and unit upper (lower)
triangular matrices and D is symmetric and block diagonal with
1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.
- **AF (input/output)**
  (N*(N+1)/2) If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by SSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by SSPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the

largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

dsptrd - reduce a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE DSPTRD( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
DOUBLE PRECISION AP(*), D(*), E(*), TAU(*)

SUBROUTINE DSPTRD_64( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
DOUBLE PRECISION AP(*), D(*), E(*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRD( UPLO, N, AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: AP, D, E, TAU

SUBROUTINE SPTRD_64( UPLO, N, AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: AP, D, E, TAU
```

C INTERFACE

#include <sunperf.h>

void dsptrd(char uplo, int n, double *ap, double *d, double *e, double *tau, int *info);

void dsptrd_64(char uplo, long n, double *ap, double *d, double *e, double *tau, long *info);

PURPOSE

dsptrd reduces a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
   Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
   H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in AP, overwriting A(1:i-1,i+1), and tau is stored in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
   Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
   H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in AP, overwriting A(i+2:n,i), and tau is stored in TAU(i).

NAME

dsptrf - compute the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE DSPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(*)

SUBROUTINE DSPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A

SUBROUTINE SPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsptrf(char uplo, int n, double *a, int *ipivot, int *info);

void dsptrf_64(char uplo, long n, double *a, long *ipivot, long *info);

PURPOSE

dsptrf computes the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

dsptri - compute the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
SUBROUTINE DSPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(*), WORK(*)

SUBROUTINE DSPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRI( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A, WORK

SUBROUTINE SPTRI_64( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void dsptri(char uplo, int n, double *a, int *ipivot, int *info);

void dsptri_64(char uplo, long n, double *a, long *ipivot, long *info);

PURPOSE

dsptri computes the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

dsptrs - solve a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
SUBROUTINE DSPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(*), B(LDB,*)

SUBROUTINE DSPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SPTRS( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE SPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dsptrs(char uplo, int n, int nrhs, double *a, int *ipivot, double *b, int ldb, int *info);

void dsptrs_64(char uplo, long n, long nrhs, double *a, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dsptrs solves a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dstebz - compute the eigenvalues of a symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE DSTEBZ( RANGE, ORDER, N, VL, VU, IL, IU, ABSTOL, D, E, M,
*      NSPLIT, W, IBLOCK, ISPLIT, WORK, IWORK, INFO)
 CHARACTER * 1 RANGE, ORDER
 INTEGER N, IL, IU, M, NSPLIT, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)

 SUBROUTINE DSTEBZ_64( RANGE, ORDER, N, VL, VU, IL, IU, ABSTOL, D, E,
*      M, NSPLIT, W, IBLOCK, ISPLIT, WORK, IWORK, INFO)
 CHARACTER * 1 RANGE, ORDER
 INTEGER*8 N, IL, IU, M, NSPLIT, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEBZ( RANGE, ORDER, [N], VL, VU, IL, IU, ABSTOL, D, E,
*       M, NSPLIT, W, IBLOCK, ISPLIT, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: RANGE, ORDER
 INTEGER :: N, IL, IU, M, NSPLIT, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEBZ_64( RANGE, ORDER, [N], VL, VU, IL, IU, ABSTOL, D,
*       E, M, NSPLIT, W, IBLOCK, ISPLIT, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: RANGE, ORDER
 INTEGER(8) :: N, IL, IU, M, NSPLIT, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void dstebz(char range, char order, int n, double vl, double vu, int il, int iu, double abstol, double *d, double *e, int *m, int *nsplit, double *w, int *iblock, int *isplit, int *info);

void dstebz_64(char range, char order, long n, double vl, double vu, long il, long iu, double abstol, double *d, double *e, long *m, long *nsplit, double *w, long *iblock, long *isplit, long *info);

PURPOSE

dstebz computes the eigenvalues of a symmetric tridiagonal matrix T. The user may ask for all eigenvalues, all eigenvalues in the half-open interval (VL, VU], or the IL-th through IU-th eigenvalues.

To avoid overflow, the matrix must be scaled so that its

largest element is no greater than overflow**(1/2) *

underflow**(1/4) in absolute value, and for greatest

accuracy, it should not be much smaller than that.

See W. Kahan ``Accurate Eigenvalues of a Symmetric Tridiagonal Matrix", Report CS41, Computer Science Dept., Stanford

University, July 21, 1966.

ARGUMENTS

- **RANGE (input)**

```
 = 'A': ("All")   all eigenvalues will be found.

 = 'V': ("Value") all eigenvalues in the half-open interval
(VL, VU] will be found.
 = 'I': ("Index") the IL-th through IU-th eigenvalues (of the
entire matrix) will be found.
```

- **ORDER (input)**

```
 = 'B': ("By Block") the eigenvalues will be grouped by
split-off block (see IBLOCK, ISPLIT) and
ordered from smallest to largest within
the block.
 = 'E': ("Entire matrix")
the eigenvalues for the entire matrix
will be ordered from smallest to
largest.
```

- **N (input)**

The order of the tridiagonal matrix T. N > = 0.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to VL, or greater than VU, will not be returned. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less. If ABSTOL is less than or equal to zero, then ULP*|T| will be used, where |T| means the 1-norm of T.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) off-diagonal elements of the tridiagonal matrix T.
- **M (output)**
  The actual number of eigenvalues found. 0 < = M < = N. (See also the description of INFO =2,3.)
- **NSPLIT (output)**
  The number of diagonal blocks in the matrix T. 1 < = NSPLIT < = N.
- **W (output)**
  On exit, the first M elements of W will contain the eigenvalues. (SSTEBZ may use the remaining N-M elements as workspace.)
- **IBLOCK (output)**
  At each row/column j where `E(j)` is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit, if INFO = 0, `IBLOCK(i)` specifies to which block (from 1 to the number of blocks) the eigenvalue `W(i)` belongs. (SSTEBZ may use the remaining N-M elements as workspace.)
- **ISPLIT (output)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc., and the NSPLIT-th consists of rows/columns ISPLIT(NSPLIT-1)+1 through `ISPLIT(NSPLIT)` =N. (Only the first NSPLIT elements will actually be used, but since the user cannot know a priori what value NSPLIT will have, N words must be reserved for ISPLIT.)
- **WORK (workspace)**
  `dimension(4*N)`
- **IWORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  some or all of the eigenvalues failed to converge or
```

were not computed:

```
 =1 or 3: Bisection failed to converge for some
eigenvalues; these eigenvalues are flagged by a
negative block number.  The effect is that the
eigenvalues may not be as accurate as the
absolute and relative tolerances.  This is
generally caused by unexpectedly inaccurate
arithmetic.
 =2 or 3: RANGE ='I' only: Not all of the eigenvalues
IL:IU were found.
```

Effect: M < IU+1-IL

Cause: non-monotonic arithmetic, causing the Sturm sequence to be non-monotonic. Cure: recalculate, using RANGE ='A', and pick

out eigenvalues IL:IU. = 4: RANGE ='I', and the Gershgorin interval initially used was too small. No eigenvalues were computed.

NAME

dstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

---

SYNOPSIS

```
 SUBROUTINE DSTEDC( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 COMPZ
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEDC_64( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 COMPZ
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEDC( COMPZ, N, D, E, Z, [LDZ], WORK, [LWORK], IWORK,
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEDC_64( COMPZ, N, D, E, Z, [LDZ], WORK, [LWORK], IWORK,
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: D, E, WORK
```

```
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstedc(char *compz, int n, double *d, double *e, double *z, int ldz, double *work, int lwork, int *iwork, int liwork, int *info);

void dstedc_64(char *compz, long n, double *d, double *e, double *z, long ldz, double *work, long lwork, long *iwork, long liwork, long *info);

PURPOSE

dstedc computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric matrix can also be found if SSYTRD or SSPTRD or SSBTRD has been used to reduce this matrix to tridiagonal form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLAED3 for details.

ARGUMENTS

- **COMPZ (input/output)**

```
 = 'N':  Compute eigenvalues only.

 = 'I':  Compute eigenvectors of tridiagonal matrix also.

 = 'V':  Compute eigenvectors of original dense symmetric
matrix also.  On entry, Z contains the orthogonal
matrix used to reduce the original matrix to
tridiagonal form.
```

- **N (input)**
  The dimension of the symmetric tridiagonal matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the orthogonal matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original symmetric matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If eigenvectors are desired, then LDZ > = max(1,N).

- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If COMPZ = 'N' or N < = 1 then LWORK must be at least 1. If COMPZ = 'V' and N > 1 then LWORK must be at least (1 + 3*N + 2*N*lg N + 3*N**2), where lg(N) = smallest integer k such that 2**k > = N. If COMPZ = 'I' and N > 1 then LWORK must be at least (1 + 4*N + N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (output)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If COMPZ = 'N' or N < = 1 then LIWORK must be at least 1. If COMPZ = 'V' and N > 1 then LIWORK must be at least (6 + 6*N + 5*N*lg N). If COMPZ = 'I' and N > 1 then LIWORK must be at least (3 + 5*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an eigenvalue while
working on the submatrix lying in rows and columns
INFO/(N+1) through mod(INFO,N+1).
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

Modified by Francoise Tisseur, University of Tennessee.

NAME

dstegr - (a) Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

---

SYNOPSIS

```
 SUBROUTINE DSTEGR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*      W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEGR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*      M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEGR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEGR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL(8) :: VL, VU, ABSTOL
REAL(8), DIMENSION(:) :: D, E, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstegr(char jobz, char range, int n, double *d, double *e, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *isuppz, int *info);

void dstegr_64(char jobz, char range, long n, double *d, double *e, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *isuppz, long *info);

PURPOSE

dstegr b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high relative accuracy by the dqds algorithm,

```
(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

Note 1 : Currently SSTEGR is only set up to find ALL the n eigenvalues and eigenvectors of T in `O(n^2)` time

Note 2 : Currently the routine SSTEIN is called when an appropriate sigma_i cannot be chosen in step (c) above. SSTEIN invokes modified Gram-Schmidt when eigenvalues are close.

Note 3 : SSTEGR works only on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs. Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the ieee standard.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

```
 = 'A': all eigenvalues will be found.

 = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix T. On exit, D is overwritten.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix T in elements 1 to N-1 of E; `E(N)` need not be set. On exit, E is overwritten.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues/eigenvectors. IF JOBZ = 'V', the eigenvalues and eigenvectors output have residual norms bounded by ABSTOL, and the dot products between different eigenvectors are bounded by ABSTOL. If ABSTOL is less than N*EPS*|T|, then N*EPS*|T| will be used in its place, where EPS is the machine precision and |T| is the 1-norm of the tridiagonal matrix. The eigenvalues are computed to an accuracy of EPS*|T| irrespective of ABSTOL. If high relative accuracy is important, set ABSTOL to DLAMCH('Safe minimum'). See Barlow and Demmel ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7 for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = `max(1,18*N)`

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**

On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= `max(1,10*N)`

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = 1, internal error in SLARRE,
if INFO  = 2, internal error in SLARRV.
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

dstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

---

SYNOPSIS

```
 SUBROUTINE DSTEIN( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 INTEGER N, M, LDZ, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEIN_64( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 INTEGER*8 N, M, LDZ, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEIN( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 INTEGER :: N, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEIN_64( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ],
*       [WORK], [IWORK], IFAIL, [INFO])
 INTEGER(8) :: N, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
 REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstein(int n, double *d, double *e, int m, double *w, int *iblock, int *isplit, double *z, int ldz, int *ifail, int *info);

void dstein_64(long n, double *d, double *e, long m, double *w, long *iblock, long *isplit, double *z, long ldz, long *ifail, long *info);

PURPOSE

dstein computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter MAXITS (currently set to 5).

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix T, in elements 1 to N-1. `E(N)` need not be set.
- **M (input)**
  The number of eigenvectors to be found. 0 < = M < = N.
- **W (input)**
  The first M elements of W contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array W from SSTEBZ with ORDER = 'B' is expected here.)
- **IBLOCK (input)**
  The submatrix indices associated with the corresponding eigenvalues in W; `IBLOCK(i)` =1 if eigenvalue `W(i)` belongs to the first submatrix from the top, =2 if `W(i)` belongs to the second submatrix, etc. (The output array IBLOCK from SSTEBZ is expected here.)
- **ISPLIT (input)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc. (The output array ISPLIT from SSTEBZ is expected here.)
- **Z (output)**
  The computed eigenvectors. The eigenvector associated with the eigenvalue `W(i)` is stored in the i-th column of Z. Any vector which fails to converge is set to its current iterate after MAXITS iterations.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK (workspace)**

```
dimension(N)
```

- **IFAIL (output)**
  On normal exit, all elements of IFAIL are zero. If one or more eigenvectors fail to converge after MAXITS iterations, then their indices are stored in array IFAIL.
- **INFO (output)**

```
 = 0: successful exit.

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, then i eigenvectors failed to converge
in MAXITS iterations.  Their indices are stored in
array IFAIL.
```

NAME

dsteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

---

SYNOPSIS

```
SUBROUTINE DSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)

SUBROUTINE DSTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEQR( COMPZ, N, D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
REAL(8), DIMENSION(:,:) :: Z

SUBROUTINE STEQR_64( COMPZ, N, D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dsteqr(char compz, int n, double *d, double *e, double *z, int ldz, int *info);

void dsteqr_64(char compz, long n, double *d, double *e, double *z, long ldz, long *info);

PURPOSE

dsteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. The eigenvectors of a full or band symmetric matrix can also be found if SSYTRD or SSPTRD or SSBTRD has been used to reduce this matrix to tridiagonal form.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvalues and eigenvectors of the original
symmetric matrix.  On entry, Z must contain the
orthogonal matrix used to reduce the original matrix
to tridiagonal form.
 = 'I':  Compute eigenvalues and eigenvectors of the
tridiagonal matrix.  Z is initialized to the identity
matrix.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the orthogonal matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original symmetric matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if eigenvectors are desired, then LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(max(1,2*N-2))` If COMPZ = 'N', then WORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm has failed to find all the eigenvalues in
a total of 30*N iterations; if INFO  = i, then i
elements of E have not converged to zero; on exit, D
and E contain the elements of a symmetric tridiagonal
matrix which is orthogonally similar to the original
matrix.
```

NAME

dsterf - compute all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm

---

SYNOPSIS

```
SUBROUTINE DSTERF( N, D, E, INFO)
INTEGER N, INFO
DOUBLE PRECISION D(*), E(*)

SUBROUTINE DSTERF_64( N, D, E, INFO)
INTEGER*8 N, INFO
DOUBLE PRECISION D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE STERF( [N], D, E, [INFO])
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: D, E

SUBROUTINE STERF_64( [N], D, E, [INFO])
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void dsterf(int n, double *d, double *e, int *info);

void dsterf_64(long n, double *d, double *e, long *info);

PURPOSE

dsterf computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm failed to find all of the eigenvalues in
a total of 30*N iterations; if INFO  = i, then i
elements of E have not converged to zero.
```

NAME

dstev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE DSTEV( JOBZ, N, DIAG, OFFD, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ
INTEGER N, LDZ, INFO
DOUBLE PRECISION DIAG(*), OFFD(*), Z(LDZ,*), WORK(*)

SUBROUTINE DSTEV_64( JOBZ, N, DIAG, OFFD, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION DIAG(*), OFFD(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEV( JOBZ, [N], DIAG, OFFD, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD, WORK
REAL(8), DIMENSION(:,:) :: Z

SUBROUTINE STEV_64( JOBZ, [N], DIAG, OFFD, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: DIAG, OFFD, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstev(char jobz, int n, double *diag, double *offd, double *z, int ldz, int *info);

void dstev_64(char jobz, long n, double *diag, double *offd, double *z, long ldz, long *info);

PURPOSE

dstev computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, if INFO = 0, the eigenvalues in ascending order.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A, stored in elements 1 to N-1 of OFFD; `OFFD(N)` need not be set, but is used by the routine. On exit, the contents of OFFD are destroyed.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with DIAG(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  If JOBZ = 'N', WORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of OFFD did not converge to zero.
```

NAME

dstevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix

---

SYNOPSIS

```
 SUBROUTINE DSTEVD( JOBZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEVD_64( JOBZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVD( JOBZ, [N], D, E, Z, [LDZ], [WORK], [LWORK], [IWORK],
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEVD_64( JOBZ, [N], D, E, Z, [LDZ], [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: D, E, WORK
 REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstevd(char jobz, int n, double *d, double *e, double *z, int ldz, int *info);

void dstevd_64(char jobz, long n, double *d, double *e, double *z, long ldz, long *info);

PURPOSE

dstevd computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A, stored in elements 1 to N-1 of E; `E(N)` need not be set, but is used by the routine. On exit, the contents of E are destroyed.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with D(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOBZ = 'N' or N < = 1 then LWORK must be at least 1. If JOBZ = 'V' and N > 1 then LWORK must be at least (1 + 4*N + N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N < = 1 then LIWORK must be at least 1. If JOBZ = 'V' and N > 1 then LIWORK must be at least 3+5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of E did not converge to zero.
```

NAME

dstevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE DSTEVR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEVR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEVR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL(8) :: VL, VU, ABSTOL
REAL(8), DIMENSION(:) :: D, E, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstevr(char jobz, char range, int n, double *d, double *e, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *isuppz, int *info);

void dstevr_64(char jobz, char range, long n, double *d, double *e, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *isuppz, long *info);

PURPOSE

dstevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, SSTEVR calls SSTEGR to compute the

eigenspectrum using Relatively Robust Representations. SSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
(a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
     is a relatively robust representation,

(b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
    relative accuracy by the dqds algorithm,

(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : SSTEVR calls SSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. SSTEVR calls SSTEBZ and SSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, D may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A in elements 1 to N-1 of E; E(N) need not be set. On exit, E may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

  If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but future releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**

The first M elements contain the selected eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 20*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal (and minimal) LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 10*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  Internal error
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA
```

NAME

dstevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

SYNOPSIS

```
 SUBROUTINE DSTEVX( JOBZ, RANGE, N, DIAG, OFFD, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION DIAG(*), OFFD(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSTEVX_64( JOBZ, RANGE, N, DIAG, OFFD, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION DIAG(*), OFFD(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVX( JOBZ, RANGE, [N], DIAG, OFFD, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: DIAG, OFFD, W, WORK
 REAL(8), DIMENSION(:,:) :: Z

 SUBROUTINE STEVX_64( JOBZ, RANGE, [N], DIAG, OFFD, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
```

```
REAL(8), DIMENSION(:) :: DIAG, OFFD, W, WORK
REAL(8), DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void dstevx(char jobz, char range, int n, double *diag, double *offd, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, double *z, int ldz, int *ifail, int *info);

void dstevx_64(char jobz, char range, long n, double *diag, double *offd, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dstevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, DIAG may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A in elements 1 to N-1 of OFFD; `OFFD(N)` need not be set. On exit, OFFD may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.

- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge (INFO > 0), then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

dstsv - compute the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix

---

SYNOPSIS

```
SUBROUTINE DSTSV( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*), B(LDB,*)

SUBROUTINE DSTSV_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE STSV( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE STSV_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dstsv(int n, int nrhs, double *l, double *d, double *subl, double *b, int ldb, int *ipiv, int *info);

void dstsv_64(long n, long nrhs, double *l, double *d, double *subl, double *b, long ldb, long *ipiv, long *info);

PURPOSE

dstsv computes the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides in B.
- **L (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.
- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.
- **SUBL (output)**

  ```
  REAL array, dimension (N)
  ```

  On exit, part of the factorization of A.
- **B (input/output)**
  The columns of B contain the right hand sides.
- **LDB (input)**
  The leading dimension of B as specified in a type or DIMENSION statement.
- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.
- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

 > 0: if INFO = i, D(k,k) is exactly zero. The factorization has
been completed, but the block diagonal matrix D is exactly singular
and division by zero will occur if it is used to solve a system of
equations.

NAME

dsttrf - compute the factorization of a symmetric tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE DSTTRF( N, L, D, SUBL, IPIV, INFO)
INTEGER N, INFO
INTEGER IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*)

SUBROUTINE DSTTRF_64( N, L, D, SUBL, IPIV, INFO)
INTEGER*8 N, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*)
```

F95 INTERFACE

```
SUBROUTINE STTRF( [N], L, D, SUBL, IPIV, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL

SUBROUTINE STTRF_64( [N], L, D, SUBL, IPIV, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL
```

C INTERFACE

#include <sunperf.h>

void dsttrf(int n, double *l, double *d, double *subl, int *ipiv, int *info);

void dsttrf_64(long n, double *l, double *d, double *subl, long *ipiv, long *info);

PURPOSE

dsttrf computes the factorization of a complex Hermitian tridiagonal matrix A.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.

- **L (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.

- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the L*D*L**H factorization of A.

- **SUBL (output)**

  ```
  REAL array, dimension (N)
  ```

  On exit, part of the factorization of A.

- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.

- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, D(k,k) is exactly zero.  The factorization has
  been completed, but the block diagonal matrix D is exactly singular
  and division by zero will occur if it is used to solve a system of
  equations.
  ```

NAME

dsttrs - computes the solution to a real system of linear equations A * X = B

SYNOPSIS

```
SUBROUTINE DSTTRS( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*), B(LDB,*)

SUBROUTINE DSTTRS_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION L(*), D(*), SUBL(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE STTRS( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE STTRS_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:) :: L, D, SUBL
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dsttrs(int n, int nrhs, double *l, double *d, double *subl, double *b, int ldb, int *ipiv, int *info);

void dsttrs_64(long n, long nrhs, double *l, double *d, double *subl, double *b, long ldb, long *ipiv, long *info);

PURPOSE

dsttrs computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric tridiagonal matrix and X and B are N-by-NRHS matrices.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.
- **NRHS (input)**

  ```
  INTEGER
  ```

  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **L (input)**

  ```
  REAL array, dimension (N-1)
  ```

  On entry, the subdiagonal elements of LL and DD.
- **D (input)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the diagonal elements of DD.
- **SUBL (input)**

  ```
  REAL array, dimension (N-2)
  ```

  On entry, the second subdiagonal elements of LL.
- **B (input)**

  ```
  REAL array, dimension
  ```

  (LDB, NRHS) On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**

  ```
  INTEGER
  ```

  The leading dimension of the array B. LDB > = max(1, N)
- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

Details of the interchanges and block pivot. If IPIV(K) > 0, 1 by 1 pivot, and if IPIV(K) = K + 1 an interchange done; If IPIV(K) < 0, 2 by 2 pivot, no interchange required.

- **INFO (output)**

```
INTEGER

= 0:  successful exit

< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

dswap - Exchange vectors x and y.

SYNOPSIS

```
SUBROUTINE DSWAP( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)

SUBROUTINE DSWAP_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SWAP( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y

SUBROUTINE SWAP_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL(8), DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void dswap(int n, double *x, int incx, double *y, int incy);

void dswap_64(long n, double *x, long incx, double *y, long incy);

PURPOSE

dswap Exchange x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input/output)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, the y vector.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, the x vector.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

dsycon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
 SUBROUTINE DSYCON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DSYCON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYCON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE SYCON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
```

```
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsycon(char uplo, int n, double *a, int lda, int *ipivot, double anorm, double *rcond, int *info);

void dsycon_64(char uplo, long n, double *a, long lda, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

dsycon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **IWORK2 (workspace)**
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dsyev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
SUBROUTINE DSYEV( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), W(*), WORK(*)

SUBROUTINE DSYEV_64( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER*8 N, LDA, LDWORK, INFO
DOUBLE PRECISION A(LDA,*), W(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYEV( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER :: N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: W, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYEV_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER(8) :: N, LDA, LDWORK, INFO
REAL(8), DIMENSION(:) :: W, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsyev(char jobz, char uplo, int n, double *a, int lda, double *w, int *info);

void dsyev_64(char jobz, char uplo, long n, double *a, long lda, double *w, long *info);

PURPOSE

dsyev computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK > = max(1,3*N-1). For optimal efficiency, LDWORK > = (NB+2)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

dsyevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
 SUBROUTINE DSYEVD( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, LDA, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION A(LDA,*), W(*), WORK(*)

 SUBROUTINE DSYEVD_64( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, LDA, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION A(LDA,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVD( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, LDA, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE SYEVD_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, LDA, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsyevd(char jobz, char uplo, int n, double *a, int lda, double *w, int *info);

void dsyevd_64(char jobz, char uplo, long n, double *a, long lda, double *w, long *info);

PURPOSE

dsyevd computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Because of large use of BLAS of level 3, SSYEVD needs N**2 more workspace than SSYEVX.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least 2*N+1. If JOBZ = 'V' and N > 1, LWORK must be at least 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK must be at least 1. If JOBZ = 'N' and N > 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

Modified by Francoise Tisseur, University of Tennessee.

NAME

dsyevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE DSYEVR( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSYEVR_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVR( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, Z

 SUBROUTINE SYEVR_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK],
*       [LIWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
INTEGER(8) :: N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL(8) :: VL, VU, ABSTOL
REAL(8), DIMENSION(:) :: W, WORK
REAL(8), DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void dsyevr(char jobz, char range, char uplo, int n, double *a, int lda, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *isuppz, int *info);

void dsyevr_64(char jobz, char range, char uplo, long n, double *a, long lda, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *isuppz, long *info);

PURPOSE

dsyevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, SSYEVR calls SSTEGR to compute the

eigenspectrum using Relatively Robust Representations. SSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
(a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
     is a relatively robust representation,

(b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
    relative accuracy by the dqds algorithm,

(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : SSYEVR calls SSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. SSYEVR calls SSTEBZ and SSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.

- **VU (input)**
  See the description of VL.

- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.

- **IU (input)**
  See the description of IL.

- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but furutre releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,26*N). For optimal efficiency, LWORK > = (NB+6)*N, where NB is the max of the blocksize for SSYTRD and SORMTR returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK > = max(1,10*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  Internal error
```

FURTHER DETAILS

Based on contributions by

Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA

NAME

dsyevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
 SUBROUTINE DSYEVX( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSYEVX_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVX( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [LDWORK], [IWORK2], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, Z

 SUBROUTINE SYEVX_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK], [LDWORK], [IWORK2],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
REAL(8) :: VL, VU, ABTOL
REAL(8), DIMENSION(:) :: W, WORK
REAL(8), DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void dsyevx(char jobz, char range, char uplo, int n, double *a, int lda, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, double *z, int ldz, int *ifail, int *info);

void dsyevx_64(char jobz, char range, char uplo, long n, double *a, long lda, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dsyevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 <= NFOUND <= N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  On normal exit, the first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,8*N). For optimal efficiency, LDWORK >= (NB+3)*N, where NB is the max of the blocksize for SSYTRD and SORMTR returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: if INFO = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.

NAME

dsygs2 - reduce a real symmetric-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE DSYGS2( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DSYGS2_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYGS2( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE SYGS2_64( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsygs2(int itype, char uplo, int n, double *a, int lda, double *b, int ldb, int *info);

void dsygs2_64(long itype, char uplo, long n, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dsygs2 reduces a real symmetric-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U')*A*inv(U)` or `inv(L)*A*inv(L')`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U` or L'*A*L.

B must have been previously factorized as U'*U or L*L' by SPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U')*A*inv(U) or inv(L)*A*inv(L');

  = 2 or 3: compute U*A*U' or L'*A*L.
  ```

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored, and how B has been factorized. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by SPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

dsygst - reduce a real symmetric-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE DSYGST( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DSYGST_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, LDA, LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYGST( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE SYGST_64( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsygst(int itype, char uplo, int n, double *a, int lda, double *b, int ldb, int *info);

void dsygst_64(long itype, char uplo, long n, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dsygst reduces a real symmetric-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**T)*A*inv(U)` or `inv(L)*A*inv(L**T)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**T or L**T*A*L.

B must have been previously factorized as U**T*U or L*L**T by SPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U**T)*A*inv(U) or inv(L)*A*inv(L**T);

  = 2 or 3: compute U*A*U**T or L**T*A*L.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored and B is factored as
U**T*U;
  = 'L':  Lower triangle of A is stored and B is factored as
L*L**T.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by SPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dsygv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSYGV( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), WORK(*)

 SUBROUTINE DSYGV_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGV( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE SYGV_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsygv(int itype, char jobz, char uplo, int n, double *a, int lda, double *b, int ldb, double *w, int *info);

void dsygv_64(long itype, char jobz, char uplo, long n, double *a, long lda, double *b, long ldb, double *w, long *info);

PURPOSE

dsygv computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also

positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **B (input/output)**
  On entry, the symmetric positive definite matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO <= N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,3*N-1). For optimal efficiency, LDWORK >= (NB+2)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEV returned an error code:

 <= N:  if INFO  = i, SSYEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

dsygvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSYGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), WORK(*)

 SUBROUTINE DSYGVD_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGVD( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE SYGVD_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, WORK
```

```
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsygvd(int itype, char jobz, char uplo, int n, double *a, int lda, double *b, int ldb, double *w, int *info);

void dsygvd_64(long itype, char jobz, char uplo, long n, double *a, long lda, double *b, long ldb, double *w, long *info);

PURPOSE

dsygvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the symmetric matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK > = 1. If JOBZ = 'N' and N > 1, LWORK > = 2*N+1. If JOBZ = 'V' and N > 1, LWORK > = 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK > = 1. If JOBZ = 'N' and N > 1, LIWORK > = 1. If JOBZ = 'V' and N > 1, LIWORK > = 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEVD returned an error code:

 < = N:  if INFO  = i, SSYEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

dsygvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE DSYGVX( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB, VL,
*      VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE DSYGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB,
*      VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION A(LDA,*), B(LDB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGVX( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B, [LDB],
*       VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK], [IWORK],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Z
```

```
 SUBROUTINE SYGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B,
*       [LDB], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Z
```

C INTERFACE

#include <sunperf.h>

void dsygvx(int itype, char jobz, char range, char uplo, int n, double *a, int lda, double *b, int ldb, double vl, double vu, int il, int iu, double abstol, int *m, double *w, double *z, int ldz, int *ifail, int *info);

void dsygvx_64(long itype, char jobz, char range, char uplo, long n, double *a, long lda, double *b, long ldb, double vl, double vu, long il, long iu, double abstol, long *m, double *w, double *z, long ldz, long *ifail, long *info);

PURPOSE

dsygvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
  ```

```
will be found.
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
 = 'U':  Upper triangle of A and B are stored;

 = 'L':  Lower triangle of A and B are stored.
```

- **N (input)**
  The order of the matrix pencil (A,B). N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the symmetric matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*DLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the

orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,8*N). For optimal efficiency, LWORK > = (NB+3)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEVX returned an error code:

 < = N:  if INFO  = i, SSYEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

dsymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE DSYMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 INTEGER M, N, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE DSYMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 INTEGER*8 M, N, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 INTEGER :: M, N, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C

 SUBROUTINE SYMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 INTEGER(8) :: M, N, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void dsymm(char side, char uplo, int m, int n, double alpha, double *a, int lda, double *b, int ldb, double beta, double *c, int ldc);

void dsymm_64(char side, char uplo, long m, long n, double alpha, double *a, long lda, double *b, long ldb, double beta, double *c, long ldc);

PURPOSE

dsymm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the symmetric matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the symmetric matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the symmetric matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the symmetric matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the symmetric matrix, such that when

UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

dsymv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE DSYMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER N, LDA, INCX, INCY
DOUBLE PRECISION ALPHA, BETA
DOUBLE PRECISION A(LDA,*), X(*), Y(*)

SUBROUTINE DSYMV_64( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INCX, INCY
DOUBLE PRECISION ALPHA, BETA
DOUBLE PRECISION A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SYMV( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INCX, INCY
REAL(8) :: ALPHA, BETA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYMV_64( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INCX, INCY
REAL(8) :: ALPHA, BETA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsymv(char uplo, int n, double alpha, double *a, int lda, double *x, int incx, double beta, double *y, int incy);

void dsymv_64(char uplo, long n, double alpha, double *a, long lda, double *x, long incx, double beta, double *y, long incy);

PURPOSE

dsymv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dsyr - perform the symmetric rank 1 operation A := alpha*x*x' + A

SYNOPSIS

```
SUBROUTINE DSYR( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
INTEGER N, INCX, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), A(LDA,*)

SUBROUTINE DSYR_64( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYR( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYR_64( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsyr(char uplo, int n, double alpha, double *x, int incx, double *a, int lda);

void dsyr_64(char uplo, long n, double alpha, double *x, long incx, double *a, long lda);

PURPOSE

dsyr performs the symmetric rank 1 operation A := alpha*x*x' + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

dsyr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

SYNOPSIS

```
SUBROUTINE DSYR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(LDA,*)

SUBROUTINE DSYR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY, LDA
DOUBLE PRECISION ALPHA
DOUBLE PRECISION X(*), Y(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY, LDA
REAL(8) :: ALPHA
REAL(8), DIMENSION(:) :: X, Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsyr2(char uplo, int n, double alpha, double *x, int incx, double *y, int incy, double *a, int lda);

void dsyr2_64(char uplo, long n, double alpha, double *x, long incx, double *y, long incy, double *a, long lda);

PURPOSE

dsyr2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

dsyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

SYNOPSIS

```
 SUBROUTINE DSYR2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER N, K, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE DSYR2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER*8 N, K, LDA, LDB, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYR2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER :: N, K, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C

 SUBROUTINE SYR2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER(8) :: N, K, LDA, LDB, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void dsyr2k(char uplo, char transa, int n, int k, double alpha, double *a, int lda, double *b, int ldb, double beta, double *c, int ldc);

void dsyr2k_64(char uplo, char transa, long n, long k, double alpha, double *a, long lda, double *b, long ldb, double beta, double *c, long ldc);

PURPOSE

dsyr2k K performs one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*B' + alpha*B*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*B + alpha*B'*A + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*A'*B + alpha*B'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'T' or 't' or 'C' or 'c', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**

On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

dsyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE DSYRFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*       X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DSYRFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*       LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYRFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*        B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE SYRFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*        IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dsyrfs(char uplo, int n, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dsyrfs_64(char uplo, long n, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dsyrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SSYTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dsyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

SYNOPSIS

```
 SUBROUTINE DSYRK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER N, K, LDA, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), C(LDC,*)

 SUBROUTINE DSYRK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER*8 N, K, LDA, LDC
 DOUBLE PRECISION ALPHA, BETA
 DOUBLE PRECISION A(LDA,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYRK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER :: N, K, LDA, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, C

 SUBROUTINE SYRK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER(8) :: N, K, LDA, LDC
 REAL(8) :: ALPHA, BETA
 REAL(8), DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void dsyrk(char uplo, char transa, int n, int k, double alpha, double *a, int lda, double beta, double *c, int ldc);

void dsyrk_64(char uplo, char transa, long n, long k, double alpha, double *a, long lda, double beta, double *c, long ldc);

PURPOSE

dsyrk performs one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*A + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*A'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'T' or 't' or 'C' or 'c', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**

On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

dsysv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DSYSV( UPLO, N, NRHS, A, LDA, IPIV, B, LDB, WORK, LWORK,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER IPIV(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE DSYSV_64( UPLO, N, NRHS, A, LDA, IPIV, B, LDB, WORK,
*      LWORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER*8 IPIV(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSV( UPLO, [N], [NRHS], A, [LDA], IPIV, B, [LDB], [WORK],
*       [LWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER, DIMENSION(:) :: IPIV
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE SYSV_64( UPLO, [N], [NRHS], A, [LDA], IPIV, B, [LDB],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDB, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIV
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsysv(char uplo, int n, int nrhs, double *a, int lda, int *ipiv, double *b, int ldb, int *info);

void dsysv_64(char uplo, long n, long nrhs, double *a, long lda, long *ipiv, double *b, long ldb, long *info);

PURPOSE

dsysv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D, as determined by SSYTRF. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a

2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of WORK. LWORK >= 1, and for best performance LWORK >= N*NB, where NB is the optimal blocksize for SSYTRF.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

dsysvx - use the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE DSYSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*       LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DSYSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*       B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*        IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*        [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE SYSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*        IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*        [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
```

```
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: FERR, BERR, WORK
REAL(8), DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void dsysvx(char fact, char uplo, int n, int nrhs, double *a, int lda, double *af, int ldaf, int *ipivot, double *b, int ldb, double *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void dsysvx_64(char fact, char uplo, long n, long nrhs, double *a, long lda, double *af, long ldaf, long *ipivot, double *b, long ldb, double *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

dsysvx uses the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',

where U (or L) is a product of permutation and unit upper (lower)
triangular matrices, and D is symmetric and block diagonal with
1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by SSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by SSYTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If

XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 3*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for SSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

dsytd2 - reduce a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE DSYTD2( UPLO, N, A, LDA, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*), D(*), E(*), TAU(*)

SUBROUTINE DSYTD2_64( UPLO, N, A, LDA, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*), D(*), E(*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE SYTD2( UPLO, [N], A, [LDA], D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:) :: D, E, TAU
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYTD2_64( UPLO, [N], A, [LDA], D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:) :: D, E, TAU
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsytd2(char uplo, int n, double *a, int lda, double *d, double *e, double *tau, int *info);

void dsytd2_64(char uplo, long n, double *a, long lda, double *d, double *e, double *tau, long *info);

PURPOSE

dsytd2 reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: Q' * A * Q = T.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in

A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

dsytf2 - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE DSYTF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
INTEGER IPIV(*)
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DSYTF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYTF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYTF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsytf2(char uplo, int n, double *a, int lda, int *ipiv, int *info);

void dsytf2_64(char uplo, long n, double *a, long lda, long *ipiv, long *info);

PURPOSE

dsytf2 computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the transpose of U, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

dsytrd - reduce a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE DSYTRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), D(*), E(*), TAU(*), WORK(*)

SUBROUTINE DSYTRD_64( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), D(*), E(*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRD( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: D, E, TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYTRD_64( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: D, E, TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsytrd(char uplo, int n, double *a, int lda, double *d, double *e, double *tau, int *info);

void dsytrd_64(char uplo, long n, double *a, long lda, double *d, double *e, double *tau, long *info);

PURPOSE

dsytrd reduces a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. For optimum performance LWORK >= N*NB, where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in

A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

dsytrf - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE DSYTRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)

SUBROUTINE DSYTRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYTRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsytrf(char uplo, int n, double *a, int lda, int *ipivot, int *info);

void dsytrf_64(char uplo, long n, double *a, long lda, long *ipivot, long *info);

PURPOSE

dsytrf computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    v    0   )   k-s

U(k)  =  (   0    I    0   )   s

         (   0    0    I   )   n-k

            k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    0     0   )  k-1

L(k)  =  (   0    I     0   )  s
```

```
(   0    v     I   )  n-k-s+1

   k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

dsytri - compute the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
SUBROUTINE DSYTRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)

SUBROUTINE DSYTRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE SYTRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dsytri(char uplo, int n, double *a, int lda, int *ipivot, int *info);

void dsytri_64(char uplo, long n, double *a, long lda, long *ipivot, long *info);

PURPOSE

dsytri computes the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

dsytrs - solve a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
SUBROUTINE DSYTRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)

SUBROUTINE DSYTRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYTRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B

SUBROUTINE SYTRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dsytrs(char uplo, int n, int nrhs, double *a, int lda, int *ipivot, double *b, int ldb, int *info);

void dsytrs_64(char uplo, long n, long nrhs, double *a, long lda, long *ipivot, double *b, long ldb, long *info);

PURPOSE

dsytrs solves a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

dtbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE DTBCON( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER N, NDIAG, LDA, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DTBCON_64( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND,
*      WORK, WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER*8 N, NDIAG, LDA, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TBCON( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER :: N, NDIAG, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE TBCON_64( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER(8) :: N, NDIAG, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
```

```
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtbcon(char norm, char uplo, char diag, int n, int ndiag, double *a, int lda, double *rcond, int *info);

void dtbcon_64(char norm, char uplo, char diag, long n, long ndiag, double *a, long lda, double *rcond, long *info);

PURPOSE

dtbcon estimates the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dtbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE DTBMV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NDIAG, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)

SUBROUTINE DTBMV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NDIAG, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TBMV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NDIAG, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TBMV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NDIAG, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtbmv(char uplo, char transa, char diag, int n, int ndiag, double *a, int lda, double *y, int incy);

void dtbmv_64(char uplo, char transa, char diag, long n, long ndiag, double *a, long lda, double *y, long incy);

PURPOSE

dtbmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The

top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE DTBRFS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DTBRFS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TBRFS( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, B, X

 SUBROUTINE TBRFS_64( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A,
*       [LDA], B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, B, X
```

C INTERFACE

#include <sunperf.h>

void dtbrfs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, double *a, int lda, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dtbrfs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, double *a, long lda, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dtbrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix.

The solution matrix X must be computed by STBTRS or some other means before entering this routine. STBRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(3*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dtbsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE DTBSV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NDIAG, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)

SUBROUTINE DTBSV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NDIAG, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TBSV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NDIAG, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TBSV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NDIAG, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtbsv(char uplo, char transa, char diag, int n, int ndiag, double *a, int lda, double *y, int incy);

void dtbsv_64(char uplo, char transa, char diag, long n, long ndiag, double *a, long lda, double *y, long incy);

PURPOSE

dtbsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper

triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtbtrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
 SUBROUTINE DTBTRS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DTBTRS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TBTRS( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE TBTRS_64( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dtbtrs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dtbtrs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dtbtrs solves a triangular system of the form

where A is a triangular band matrix of order N, and B is an N-by NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of A. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd). If DIAG = 'U', the diagonal elements of A

are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.
```

NAME

dtgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B)

SYNOPSIS

```
 SUBROUTINE DTGEVC( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE DTGEVC_64( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEVC( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE TGEVC_64( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dtgevc(char side, char howmny, logical *select, int n, double *a, int lda, double *b, int ldb, double *vl, int ldvl, double *vr, int ldvr, int mm, int *m, int *info);

void dtgevc_64(char side, char howmny, logical *select, long n, double *a, long lda, double *b, long ldb, double *vl, long ldvl, double *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

dtgevc computes some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B).

The right generalized eigenvector x and the left generalized eigenvector y of (A,B) corresponding to a generalized eigenvalue w are defined by:

```
(A - wB) * x = 0  and  y**H * (A - wB) = 0
```

where y**H denotes the conjugate tranpose of y.

If an eigenvalue w is determined by zero diagonal elements of both A and B, a unit vector is returned as the corresponding eigenvector.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of (A,B), or the products Z*X and/or Q*Y, where Z and Q are input orthogonal matrices. If (A,B) was obtained from the generalized real-Schur factorization of an original pair of matrices

```
(A0,B0) = (Q*A*Z**H,Q*B*Z**H),
```

then Z*X and Q*Y are the matrices of right or left eigenvectors of A.

A must be block upper triangular, with 1-by-1 and 2-by-2 diagonal blocks. Corresponding to each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one

eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R': compute right eigenvectors only;

  = 'L': compute left eigenvectors only;

  = 'B': compute both right and left eigenvectors.
  ```
- **HOWMNY (input)**

```
 = 'A': compute all right and/or left eigenvectors;

 = 'B': compute all right and/or left eigenvectors, and
backtransform them using the input matrices supplied
in VR and/or VL;
 = 'S': compute selected right and/or left eigenvectors,
specified by the logical array SELECT.
```

- **SELECT (input)**
  If HOWMNY ='S', SELECT specifies the eigenvectors to be computed. If HOWMNY ='A' or 'B', SELECT is not referenced. To select the real eigenvector corresponding to the real eigenvalue w(j), `SELECT(j)` must be set to .TRUE. To select the complex eigenvector corresponding to a complex conjugate pair `w(j)` and w(j+1), either `SELECT(j)` or `SELECT(j+1)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A.
- **LDA (input)**
  The leading dimension of array A. LDA >= max(1, N).
- **B (input)**
  The upper triangular matrix B. If A has a 2-by-2 diagonal block, then the corresponding 2-by-2 block of B must be diagonal with positive elements.
- **LDB (input)**
  The leading dimension of array B. LDB >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the orthogonal matrix Q of left Schur vectors returned by SHGEQZ). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of (A,B); if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.

  A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.
- **LDVL (input)**
  The leading dimension of array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the orthogonal matrix Z of right Schur vectors returned by SHGEQZ). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of (A,B); if HOWMNY = 'B', the matrix Z*X; if HOWMNY = 'S', the right eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.

  A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the 2-by-2 block (INFO:INFO+1) does not have a complex
eigenvalue.
```

FURTHER DETAILS

Allocation of workspace:

---------- -- ---------

```
WORK( j )  = 1-norm of j-th column of A, above the diagonal
WORK( N+j )  = 1-norm of j-th column of B, above the diagonal
WORK( 2*N+1:3*N )  = real part of eigenvector

WORK( 3*N+1:4*N )  = imaginary part of eigenvector

WORK( 4*N+1:5*N )  = real part of back-transformed eigenvector
WORK( 5*N+1:6*N )  = imaginary part of back-transformed eigenvector
```

Rowwise vs. columnwise solution methods:

------- -- ---------- -------- -------

Finding a generalized eigenvector consists basically of solving the singular triangular system

```
 (A - w B) x  = 0     (for right) or:   (A - w B)**H y  = 0  (for left)
```

Consider finding the i-th right eigenvector (assume all eigenvalues are real). The equation to be solved is: i

0 = sum `C(j,k)` `v(k)` = sum `C(j,k)` `v(k)` for j = i,. . .,1 k =j k =j

where C = (A - w B) (The components `v(i+1:n)` are 0.)

The ``rowwise'' method is:

(1) `v(i)` : = 1

for j = i-1,. . .,1:

```
                       i

    (2) compute  s  = - sum C(j,k) v(k)   and

                      k =j+1

    (3) v(j) : = s / C(j,j)
```

Step 2 is sometimes called the ``dot product'' step, since it is an inner product between the j-th row and the portion of the eigenvector that has been computed so far.

The ``columnwise'' method consists basically in doing the sums for all the rows in parallel. As each `v(j)` is computed, the contribution of `v(j)` times the j-th column of C is added to the partial sums. Since FORTRAN arrays are stored columnwise, this has the advantage that at each step, the elements of C that are accessed are adjacent to one another, whereas with the rowwise method, the elements accessed at a step are spaced LDA (and LDB) words apart.

When finding left eigenvectors, the matrix in question is the transpose of the one in storage, so the rowwise method then actually accesses columns of A and B at each step, and so is the preferred method.

NAME

dtgexc - reorder the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation (A, B) = Q * (A, B) * Z',

---

SYNOPSIS

```
 SUBROUTINE DTGEXC( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z, LDZ,
*       IFST, ILST, WORK, LWORK, INFO)
 INTEGER N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL WANTQ, WANTZ
 DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)

 SUBROUTINE DTGEXC_64( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z,
*       LDZ, IFST, ILST, WORK, LWORK, INFO)
 INTEGER*8 N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL*8 WANTQ, WANTZ
 DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEXC( WANTQ, WANTZ, N, A, [LDA], B, [LDB], Q, [LDQ], Z,
*        [LDZ], IFST, ILST, [WORK], [LWORK], [INFO])
 INTEGER :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL :: WANTQ, WANTZ
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE TGEXC_64( WANTQ, WANTZ, N, A, [LDA], B, [LDB], Q, [LDQ],
*        Z, [LDZ], IFST, ILST, [WORK], [LWORK], [INFO])
 INTEGER(8) :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL(8) :: WANTQ, WANTZ
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dtgexc(logical wantq, logical wantz, int n, double *a, int lda, double *b, int ldb, double *q, int ldq, double *z, int ldz, int *ifst, int *ilst, int *info);

void dtgexc_64(logical wantq, logical wantz, long n, double *a, long lda, double *b, long ldb, double *q, long ldq, double *z, long ldz, long *ifst, long *ilst, long *info);

PURPOSE

dtgexc reorders the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation

so that the diagonal block of (A, B) with row index IFST is moved to row ILST.

(A, B) must be in generalized real Schur canonical form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

Optionally, the matrices Q and Z of generalized Schur vectors are updated.

```
Q(in) * A(in) * Z(in)' = Q(out) * A(out) * Z(out)'
Q(in) * B(in) * Z(in)' = Q(out) * B(out) * Z(out)'
```

ARGUMENTS

- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the matrix A in generalized real Schur canonical form. On exit, the updated matrix A, again in generalized real Schur canonical form.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the matrix B in generalized real Schur canonical form (A,B). On exit, the updated matrix B, again in generalized real Schur canonical form (A,B).
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **Q (input/output)**

On entry, if WANTQ = .TRUE., the orthogonal matrix Q. On exit, the updated matrix Q. If WANTQ = .FALSE., Q is not referenced.

- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If WANTQ = .TRUE., LDQ > = N.
- **Z (input/output)**
  On entry, if WANTZ = .TRUE., the orthogonal matrix Z. On exit, the updated matrix Z. If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If WANTZ = .TRUE., LDZ > = N.
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of (A, B). The block with row index IFST is moved to row ILST, by a sequence of swapping between adjacent blocks. On exit, if IFST pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; ILST always points to the first row of the block in its final position (which may differ from its input value by +1 or -1). 1 < = IFST, ILST < = N.
- **ILST (input/output)**
  See the description of IFST.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 4*N + 16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 =0:  successful exit.

 <0:  if INFO  = -i, the i-th argument had an illegal value.

 =1:  The transformed matrix pair (A, B) would be too far
from generalized Schur form; the problem is ill-
conditioned. (A, B) may have been partially reordered,
and ILST points to the first row of the current
position of the block being moved.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

NAME

dtgsen - reorder the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B

SYNOPSIS

```
 SUBROUTINE DTGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 INTEGER IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL WANTQ, WANTZ
 LOGICAL SELECT(*)
 DOUBLE PRECISION PL, PR
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), DIF(*), WORK(*)

 SUBROUTINE DTGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 INTEGER*8 IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 WANTQ, WANTZ
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION PL, PR
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), DIF(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B, [LDB],
*       ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
 INTEGER :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL :: WANTQ, WANTZ
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8) :: PL, PR
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, DIF, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE TGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF,
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 INTEGER(8) :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8) :: WANTQ, WANTZ
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8) :: PL, PR
 REAL(8), DIMENSION(:) :: ALPHAR, ALPHAI, BETA, DIF, WORK
 REAL(8), DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void dtgsen(int ijob, logical wantq, logical wantz, logical *select, int n, double *a, int lda, double *b, int ldb, double *alphar, double *alphai, double *beta, double *q, int ldq, double *z, int ldz, int *m, double *pl, double *pr, double *dif, int *info);

void dtgsen_64(long ijob, logical wantq, logical wantz, logical *select, long n, double *a, long lda, double *b, long ldb, double *alphar, double *alphai, double *beta, double *q, long ldq, double *z, long ldz, long *m, double *pl, double *pr, double *dif, long *info);

PURPOSE

dtgsen reorders the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B. The leading columns of Q and Z form orthonormal bases of the corresponding left and right eigen- spaces (deflating subspaces). (A, B) must be in generalized real Schur canonical form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

DTGSEN also computes the generalized eigenvalues

```
w(j) = (ALPHAR(j) + i*ALPHAI(j))/BETA(j)
```

of the reordered matrix pair (A, B).

Optionally, DTGSEN computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[(A11,B11), (A22,B22)] and Difl[(A11,B11), (A22,B22)], i.e. the `separation(s)` between the matrix pairs (A11, B11) and (A22,B22) that correspond to the selected cluster and the eigenvalues outside the cluster, resp., and norms of ``projections'' onto left and right eigenspaces w.r.t. the selected cluster in the (1,1)-block.

ARGUMENTS

- **IJOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (PL and PR) or the deflating subspaces (Difu and Difl):

  ```
  =0: Only reorder w.r.t. SELECT. No extras.

  =1: Reciprocal of norms of "projections" onto left and right
eigenspaces w.r.t. the selected cluster (PL and PR).
  =2: Upper bounds on Difu and Difl. F-norm-based estimate
  ```

  (DIF(1:2)).

  ```
  =3: Estimate of Difu and Difl. 1-norm-based estimate
  ```

  (DIF(1:2)). About 5 times as expensive as IJOB = 2. =4: Compute PL, PR and DIF (i.e. 0, 1 and 2 above): Economic version to get it all. =5: Compute PL, PR and DIF (i.e. 0, 1 and 3 above)
- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select a complex conjugate pair of eigenvalues `w(j)` and w(j+1), corresponding to a 2-by-2 diagonal block, either `SELECT(j)` or `SELECT(j+1)` or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper quasi-triangular matrix A, with (A, B) in generalized real Schur canonical form. On exit, A is overwritten by the reordered matrix A.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **B (input/output)**
  On entry, the upper triangular matrix B, with (A, B) in generalized real Schur canonical form. On exit, B is overwritten by the reordered matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A,B) were further reduced to triangular form using complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., Q is an N-by-N matrix. On exit, Q has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B); The leading M columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; and if WANTQ = .TRUE., LDQ >= N.
- **Z (input/output)**
  On entry, if WANTZ = .TRUE., Z is an N-by-N matrix. On exit, Z has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B); The leading M columns of Z form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1; If WANTZ = .TRUE., LDZ >= N.
- **M (output)**
  The dimension of the specified pair of left and right eigen- spaces (deflating subspaces). 0 <= M <= N.
- **PL (output)**
  If IJOB = 1, 4 or 5, PL, PR are lower bounds on the reciprocal of the norm of ``projections'' onto left and right eigenspaces with respect to the selected cluster. 0 < PL, PR <= 1. If M = 0 or M = N, PL = PR = 1. If IJOB = 0, 2 or 3, PL and PR are not referenced.
- **PR (output)**
  See the description of PL.
- **DIF (output)**
  If IJOB >= 2, `DIF(1:2)` store the estimates of Difu and Difl.

  If IJOB = 2 or 4, `DIF(1:2)` are F-norm-based upper bounds on

  Difu and Difl. If IJOB = 3 or 5, `DIF(1:2)` are 1-norm-based estimates of Difu and Difl. If M = 0 or N, `DIF(1:2)` = F-norm([A, B]). If IJOB = 0 or 1, DIF is not referenced.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 4*N+16. If IJOB = 1, 2 or 4, LWORK >= MAX(4*N+16, 2*M*(N-M)). If IJOB = 3 or 5, LWORK >= MAX(4*N+16, 4*M*(N-M)).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced. Otherwise, on exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 1. If IJOB = 1, 2 or 4, LIWORK >= N+6. If IJOB = 3 or 5, LIWORK >= MAX(2*M*(N-M), N+6).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 =0: Successful exit.

 <0: If INFO  = -i, the i-th argument had an illegal value.

 =1: Reordering of (A, B) failed because the transformed
matrix pair (A, B) would be too far from generalized
Schur form; the problem is very ill-conditioned.
(A, B) may have been partially reordered.
```

```
If requested, 0 is returned in DIF(*), PL and PR.
```

FURTHER DETAILS

DTGSEN first collects the selected eigenvalues by computing orthogonal U and W that move them to the top left corner of (A, B). In other words, the selected eigenvalues are the eigenvalues of (A11, B11) in:

```
U'*(A, B)*W  = (A11 A12) (B11 B12) n1
               ( 0  A22),( 0  B22) n2
                 n1  n2    n1  n2
```

where N = n1+n2 and U' means the transpose of U. The first n1 columns of U and W span the specified pair of left and right eigenspaces (deflating subspaces) of (A, B).

If (A, B) has been obtained from the generalized real Schur decomposition of a matrix pair (C, D) = Q*(A, B)*Z', then the reordered generalized real Schur form of (C, D) is given by

```
(C, D)  = (Q*U)*(U'*(A, B)*W)*(Z*W)',
```

and the first n1 columns of Q*U and Z*W span the corresponding deflating subspaces of (C, D) (Q and Z store Q*U and Z*W, resp.).

Note that if the selected eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

The reciprocal condition numbers of the left and right eigenspaces spanned by the first n1 columns of U and W (or Q*U and Z*W) may be returned in DIF(1:2), corresponding to Difu and Difl, resp.

The Difu and Difl are defined as:

ifu[(A11, B11), (A22, B22)] = sigma-min(Zu)

and ifl[(A11, B11), (A22, B22)] = Difu[(A22, B22), (A11, B11)],

where sigma-min(Zu) is the smallest singular value of the (2*n1*n2)-by-(2*n1*n2) matrix

u = [kron(In2, A11) -kron(A22', In1)]

```
    [ kron(In2, B11)  -kron(B22', In1) ].
```

Here, Inx is the identity matrix of size nx and A22' is the transpose of A22. kron(X, Y) is the Kronecker product between the matrices X and Y.

When DIF(2) is small, small changes in (A, B) can cause large changes in the deflating subspace. An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is PS * norm((A, B)) / DIF(2),

where EPS is the machine precision.

The reciprocal norm of the projectors on the left and right eigenspaces associated with (A11, B11) may be returned in PL and PR. They are computed as follows. First we compute L and R so that P*(A, B)*Q is block diagonal, where

```
= ( I -L ) n1           Q  = ( I R ) n1
  ( 0  I ) n2    and         ( 0 I ) n2
    n1 n2                     n1 n2
```

and (L, R) is the solution to the generalized Sylvester equation 11*R - L*A22 = -A12 11*R - L*B22 = -B12

Then PL = (F-norm(L)**2+1)**(-1/2) and PR = (F-norm(R)**2+1)**(-1/2). An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

PS * norm((A, B)) / PL.

There are also global error bounds which valid for perturbations up to a certain restriction: A lower bound (x) on the smallest F-norm(E,F) for which an eigenvalue of (A11, B11) may move and coalesce with an eigenvalue of (A22, B22) under perturbation (E,F), (i.e. (A + E, B + F), is

```
x  = min(Difu,Difl)/((1/(PL*PL)+1/(PR*PR))**(1/2)+2*max(1/PL,1/PR)).
```

An approximate bound on x can be computed from DIF(1:2), PL and PR.

If y = (F-norm(E,F) / x) < = 1, the angles between the perturbed (L', R') and unperturbed (L, R) left and right deflating subspaces associated with the selected cluster in the (1,1)-blocks can be bounded as

```
max-angle(L, L')  < = arctan( y * PL / (1 - y * (1 - PL * PL)**(1/2))
max-angle(R, R')  < = arctan( y * PR / (1 - y * (1 - PR * PR)**(1/2))
```

See LAPACK User's Guide section 4.11 or the following references for more information.

Note that if the default method for computing the Frobenius-norm- based estimate DIF is not wanted (see SLATDF), then the parameter IDIFJB (see below) should be changed from 3 to 4 (routine SLATDF (IJOB = 2 will be used)). See STGSYL for more details.

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

References

```
 = = = = = = = = = =
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software,

```
    Report UMINF - 94.04, Department of Computing Science, Umea
    University, S-901 87 Umea, Sweden, 1994. Also as LAPACK Working
    Note 87. To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

dtgsja - compute the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B

SYNOPSIS

```
 SUBROUTINE DTGSJA( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B, LDB,
*      TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, NCYCLE,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)

 SUBROUTINE DTGSJA_64( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B,
*      LDB, TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK,
*      NCYCLE, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSJA( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE TGSJA_64( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER(8) :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK
 REAL(8), DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void dtgsja(char jobu, char jobv, char jobq, int m, int p, int n, int k, int l, double *a, int lda, double *b, int ldb, double tola, double tolb, double *alpha, double *beta, double *u, int ldu, double *v, int ldv, double *q, int ldq, int *ncycle, int *info);

void dtgsja_64(char jobu, char jobv, char jobq, long m, long p, long n, long k, long l, double *a, long lda, double *b, long ldb, double tola, double tolb, double *alpha, double *beta, double *u, long ldu, double *v, long ldv, double *q, long ldq, long *ncycle, long *info);

PURPOSE

dtgsja computes the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B.

On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine SGGSVP from a general M-by-N matrix A and P-by-N matrix B:

```
             N-K-L  K    L

A =    K ( 0    A12  A13 ) if M-K-L >= 0;

       L ( 0     0   A23 )

   M-K-L ( 0     0    0  )

            N-K-L  K    L

A =  K ( 0    A12  A13 ) if M-K-L < 0;

   M-K ( 0     0   A23 )

            N-K-L  K    L

B =  L ( 0     0   B13 )

   P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal.

On exit,

```
            U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R ),
```

where U, V and Q are orthogonal matrices, Z' denotes the transpose of Z, R is a nonsingular upper triangular matrix, and D1 and D2 are ``diagonal'' matrices, which are of the following structures:

If M-K-L >= 0,

```
                    K  L

       D1 =     K ( I  0 )

                L ( 0  C )

             M-K-L ( 0  0 )
```

```
                  K  L
       D2 = L   ( 0  S )
            P-L ( 0  0 )
               N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 ) K
            L (  0    0   R22 ) L
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),
  S = diag( BETA(K+1),  ... , BETA(K+L) ),
  C**2 + S**2 = I.
  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                K M-K K+L-M
    D1 =   K ( I  0    0   )
         M-K ( 0  C    0   )
                  K M-K K+L-M
    D2 =   M-K ( 0  S    0   )
         K+L-M ( 0  0    I   )
           P-L ( 0  0    0   )
                N-K-L  K   M-K  K+L-M
          M-K ( 0     0   R22  R23  )
        K+L-M ( 0     0    0   R33  )
```

where

C = diag(ALPHA(K+1), ... , ALPHA(M)),

S = diag(BETA(K+1), ... , BETA(M)),

C**2 + S**2 = I.

R = (R11 R12 R13) is stored in A(1:M, N-K-L+1:N) and R33 is stored (0 R22 R23)

in B(M-K+1:L,N+M-K-L+1:N) on exit.

The computation of the orthogonal transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U1, V1, or Q1.

STGSJA essentially uses a variant of Kogbetliantz algorithm to reduce min(L,M-K)-by-L triangular (or trapezoidal) matrix A23 and L-by-L

matrix B13 to the form:

```
U1'*A13*Q1 = C1*R1; V1'*B13*Q1 = S1*R1,
```

where U1, V1 and Q1 are orthogonal matrix, and Z' is the transpose of Z. C1 and S1 are diagonal matrices satisfying

```
C1**2 + S1**2 = I,
```

and R1 is an L-by-L nonsingular upper triangular matrix.

ARGUMENTS

- **JOBU (input)**

```
 = 'U':  U must contain an orthogonal matrix U1 on entry, and
the product U1*U is returned;
 = 'I':  U is initialized to the unit matrix, and the
orthogonal matrix U is returned;
 = 'N':  U is not computed.
```

- **JOBV (input)**

```
 = 'V':  V must contain an orthogonal matrix V1 on entry, and
the product V1*V is returned;
 = 'I':  V is initialized to the unit matrix, and the
orthogonal matrix V is returned;
 = 'N':  V is not computed.
```

- **JOBQ (input)**

```
 = 'Q':  Q must contain an orthogonal matrix Q1 on entry, and
the product Q1*Q is returned;
 = 'I':  Q is initialized to the unit matrix, and the
orthogonal matrix Q is returned;
 = 'N':  Q is not computed.
```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **K (input)**
  K and L specify the subblocks in the input matrices A and B:

  A23 = `A(K+1:MIN(K+L,M),N-L+1:N)` and B13 = `B(1:L,N-L+1:N)` of A and B, whose GSVD is going to be computed by STGSJA. See Further details.
- **L (input)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, `A(N-K+1:N,1:MIN(K+L,M)`) contains the triangular matrix R or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, if necessary, `B(M-K+1:L,N+M-K-L+1:N)` contains a part of R. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the convergence criteria for the Jacobi- Kogbetliantz iteration procedure. Generally, they are the same as used

in the preprocessing step, say TOLA = max(M,N)*norm(A)*MACHEPS, TOLB = max(P,N)*norm(B)*MACHEPS.

- **TOLB (input)**
  See the description of TOLA.
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; `ALPHA(1:K)` = 1,

  `BETA(1:K)` = 0, and if M-K-L >= 0, `ALPHA(K+1:K+L)` = diag(C),

  `BETA(K+1:K+L)` = diag(S), or if M-K-L < 0, `ALPHA(K+1:M)` = C, `ALPHA(M+1:K+L)` = 0

  `BETA(K+1:M)` = S, `BETA(M+1:K+L)` = 1. Furthermore, if K+L < N, `ALPHA(K+L+1:N)` = 0 and

  `BETA(K+L+1:N)` = 0.
- **BETA (output)**
  See the description of ALPHA.
- **U (input/output)**
  On entry, if JOBU = 'U', U must contain a matrix U1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBU = 'I', U contains the orthogonal matrix U; if JOBU = 'U', U contains the product U1*U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (input/output)**
  On entry, if JOBV = 'V', V must contain a matrix V1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBV = 'I', V contains the orthogonal matrix V; if JOBV = 'V', V contains the product V1*V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.
- **Q (input/output)**
  On entry, if JOBQ = 'Q', Q must contain a matrix Q1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBQ = 'I', Q contains the orthogonal matrix Q; if JOBQ = 'Q', Q contains the product Q1*Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **NCYCLE (output)**
  The number of cycles required for convergence.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.

= 1:  the procedure does not converge after MAXIT cycles.
```

NAME

dtgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z

SYNOPSIS

```
 SUBROUTINE DTGSNA( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL, LDVL,
*      VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), S(*), DIF(*), WORK(*)

 SUBROUTINE DTGSNA_64( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), S(*), DIF(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSNA( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, DIF, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE TGSNA_64( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB],
```

```
*       VL, [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, DIF, WORK
 REAL(8), DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dtgsna(char job, char howmnt, logical *select, int n, double *a, int lda, double *b, int ldb, double *vl, int ldvl, double *vr, int ldvr, double *s, double *dif, int mm, int *m, int *info);

void dtgsna_64(char job, char howmnt, logical *select, long n, double *a, long lda, double *b, long ldb, double *vl, long ldvl, double *vr, long ldvr, double *s, double *dif, long mm, long *m, long *info);

PURPOSE

dtgsna estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z.

(A, B) must be in generalized real Schur form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (DIF):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (DIF);

  = 'B': for both eigenvalues and eigenvectors (S and DIF).
  ```
- **HOWMNT (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNT = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the eigenpair corresponding to a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select condition numbers corresponding to a complex conjugate pair of eigenvalues `w(j)` and w(j+1), either `SELECT(j)`

or SELECT(j+1) or both, must be set to .TRUE.. If HOWMNT = 'A', SELECT is not referenced.

- **N (input)**
  The order of the square matrix pair (A, B). N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A in the pair (A,B).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The upper triangular matrix B in the pair (A,B).
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by STGEVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1. If JOB = 'E' or 'B', LDVL >= N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns ov VR, as returned by STGEVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1. If JOB = 'E' or 'B', LDVR >= N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of S are set to the same value. Thus S(j), DIF(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **DIF (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of DIF are set to the same value. If the eigenvalues cannot be reordered to compute DIF(j), DIF(j) is set to 0; this can only occur when the true value would be very small anyway. If JOB = 'E', DIF is not referenced.
- **MM (input)**
  The number of elements in the arrays S and DIF. MM >= M.
- **M (output)**
  The number of elements of the arrays S and DIF used to store the specified condition numbers; for each selected real eigenvalue one element is used, and for each selected complex conjugate pair of eigenvalues, two elements are used. If HOWMNT = 'A', M is set to N.
- **WORK (workspace)**
  If JOB = 'E', WORK is not referenced. Otherwise, on exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= N. If JOB = 'V' or 'B' LWORK >= 2*N*(N+2)+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  dimension(N+6) If JOB = 'E', IWORK is not referenced.
- **INFO (output)**

  =0: Successful exit

```
      <0: If INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of a generalized eigenvalue w = (a, b) is defined as

(w) = (|u'Av|**2 + |u'Bv|**2)**(1/2) / (norm(u)*norm(v))

where u and v are the left and right eigenvectors of (A, B) corresponding to w; |z| denotes the absolute value of the complex number, and `norm(u)` denotes the 2-norm of the vector u.

The pair (a, b) corresponds to an eigenvalue w = a/b (= u'Av/u'Bv) of the matrix pair (A, B). If both a and b equal zero, then (A B) is singular and `S(I)` = -1 is returned.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(A, B) / `S(I)`

where EPS is the machine precision.

The reciprocal of the condition number `DIF(i)` of right eigenvector u and left eigenvector v corresponding to the generalized eigenvalue w is defined as follows:

a) If the i-th eigenvalue w = (a,b) is real

```
   Suppose U and V are orthogonal transformations such that

              U'*(A, B)*V   = (S, T)  = ( a    *  ) ( b  *  )  1
                                        ( 0  S22 ),( 0 T22 )  n-1
                                          1  n-1     1 n-1

   Then the reciprocal condition number DIF(i) is

              Difl((a, b), (S22, T22))  = sigma-min( Zl ),

   where sigma-min(Zl) denotes the smallest singular value of the
   2(n-1)-by-2(n-1) matrix

       Zl  = [ kron(a, In-1)  -kron(1, S22) ]

             [ kron(b, In-1)  -kron(1, T22) ] .

   Here In-1 is the identity matrix of size n-1. kron(X, Y) is the
   Kronecker product between the matrices X and Y.

   Note that if the default method for computing DIF(i) is wanted
   (see SLATDF), then the parameter DIFDRI (see below) should be
   changed from 3 to 4 (routine SLATDF(IJOB  = 2 will be used)).
   See STGSYL for more details.
```

b) If the i-th and (i+1)-th eigenvalues are complex conjugate pair,

```
Suppose U and V are orthogonal transformations such that

           U'*(A, B)*V  = (S, T)  = ( S11  *   ) ( T11  *  )  2
                                  ( 0    S22 ),( 0    T22) n-2
                                    2    n-2     2    n-2

and (S11, T11) corresponds to the complex conjugate eigenvalue
pair (w, conjg(w)). There exist unitary matrices U1 and V1 such
that

    U1'*S11*V1  = ( s11 s12 )   and U1'*T11*V1  = ( t11 t12 )
                  (  0  s22 )                     (  0  t22 )

where the generalized eigenvalues w  = s11/t11 and

conjg(w)  = s22/t22.

Then the reciprocal condition number DIF(i) is bounded by

    min( d1, max( 1, |real(s11)/real(s22)| )*d2 )

where, d1  = Difl((s11, t11), (s22, t22))  = sigma-min(Z1), where
Z1 is the complex 2-by-2 matrix

         Z1  =  [ s11  -s22 ]

                [ t11  -t22 ],

This is done by computing (using real arithmetic) the

roots of the characteristical polynomial det(Z1' * Z1 - lambda I),
where Z1' denotes the conjugate transpose of Z1 and det(X) denotes
the determinant of X.

and d2 is an upper bound on Difl((S11, T11), (S22, T22)), i.e. an
upper bound on sigma-min(Z2), where Z2 is (2n-2)-by-(2n-2)

         Z2  = [ kron(S11', In-2)  -kron(I2, S22) ]

               [ kron(T11', In-2)  -kron(I2, T22) ]

Note that if the default method for computing DIF is wanted (see
SLATDF), then the parameter DIFDRI (see below) should be changed
from 3 to 4 (routine SLATDF(IJOB  = 2 will be used)). See STGSYL
for more details.
```

For each eigenvalue/vector specified by SELECT, DIF stores a Frobenius norm-based estimate of Difl.

An approximate error bound for the i-th computed eigenvector VL(i) or VR(i) is given by

```
          EPS * norm(A, B) / DIF(i).
```

See ref. [2-3] for more details and further references.

Based on contributions by

Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.

References

= = = = = = = = = =

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software,

Report UMINF - 94.04, Department of Computing Science, Umea
University, S-901 87 Umea, Sweden, 1994. Also as LAPACK Working
Note 87. To appear in Numerical Algorithms, 1996.

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

dtgsyl - solve the generalized Sylvester equation

SYNOPSIS

```
 SUBROUTINE DTGSYL( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION SCALE, DIF
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)

 SUBROUTINE DTGSYL_64( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER*8 IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION SCALE, DIF
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSYL( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C, [LDC],
*       D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: TRANS
 INTEGER :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: SCALE, DIF
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A, B, C, D, E, F

 SUBROUTINE TGSYL_64( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C,
*       [LDC], D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK],
*       [IWORK], [INFO])
 CHARACTER(LEN=1) :: TRANS
 INTEGER(8) :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8) :: SCALE, DIF
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A, B, C, D, E, F
```

C INTERFACE

#include <sunperf.h>

void dtgsyl(char trans, int ijob, int m, int n, double *a, int lda, double *b, int ldb, double *c, int ldc, double *d, int ldd, double *e, int lde, double *f, int ldf, double *scale, double *dif, int *info);

void dtgsyl_64(char trans, long ijob, long m, long n, double *a, long lda, double *b, long ldb, double *c, long ldc, double *d, long ldd, double *e, long lde, double *f, long ldf, double *scale, double *dif, long *info);

PURPOSE

dtgsyl solves the generalized Sylvester equation:

```
A * R - L * B = scale * C                (1)
D * R - L * E = scale * F
```

where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively, with real entries. (A, D) and (B, E) must be in generalized (real) Schur canonical form, i.e. A, B are upper quasi triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F). 0 <= SCALE <= 1 is an output scaling factor chosen to avoid overflow.

In matrix notation (1) is equivalent to solve Zx = scale b, where Z is defined as

```
Z = [ kron(In, A)  -kron(B', Im) ]         (2)
    [ kron(In, D)  -kron(E', Im) ].
```

Here Ik is the identity matrix of size k and X' is the transpose of X. kron(X, Y) is the Kronecker product between the matrices X and Y.

If TRANS = 'T', STGSYL solves the transposed system Z'*y = scale*b, which is equivalent to solve for R and L in

```
A' * R  + D' * L   = scale *  C          (3)
R  * B' + L  * E'  = scale * (-F)
```

This case (TRANS = 'T') is used to compute an one-norm-based estimate of Dif[(A,D), (B,E)], the separation between the matrix pairs (A,D) and (B,E), using SLACON.

If IJOB >= 1, STGSYL computes a Frobenius norm-based estimate of Dif[(A,D),(B,E)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z. See [1-2] for more information.

This is a level 3 BLAS algorithm.

ARGUMENTS

- **TRANS (input)**

```
 = 'N', solve the generalized Sylvester equation (1).
 = 'T', solve the 'transposed' system (3).
```

- **IJOB (input)**
  Specifies what kind of functionality to be performed. =0: solve (1) only.

```
 =1: The functionality of 0 and 3.

 =2: The functionality of 0 and 4.

 =3: Only an estimate of Dif[(A,D), (B,E)] is computed.
(look ahead strategy IJOB   = 1 is used).
 =4: Only an estimate of Dif[(A,D), (B,E)] is computed.
( SGECON on sub-systems is used ).
Not referenced if TRANS  = 'T'.
```

- **M (input)**
  The order of the matrices A and D, and the row dimension of the matrices C, F, R and L.
- **N (input)**
  The order of the matrices B and E, and the column dimension of the matrices C, F, R and L.
- **A (input)**
  The upper quasi triangular matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, M).
- **B (input)**
  The upper quasi triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **C (input/output)**
  On entry, C contains the right-hand-side of the first matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, C has been overwritten by the solution R. If IJOB = 3 or 4 and TRANS = 'N', C holds R, the solution achieved during the computation of the Dif-estimate.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1, M).
- **D (input)**
  The upper triangular matrix D.
- **LDD (input)**
  The leading dimension of the array D. LDD > = max(1, M).
- **E (input)**
  The upper triangular matrix E.
- **LDE (input)**
  The leading dimension of the array E. LDE > = max(1, N).
- **F (input/output)**
  On entry, F contains the right-hand-side of the second matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, F has been overwritten by the solution L. If IJOB = 3 or 4 and TRANS = 'N', F holds L, the solution achieved during the computation of the Dif-estimate.
- **LDF (input)**
  The leading dimension of the array F. LDF > = max(1, M).
- **SCALE (output)**
  On exit SCALE is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. SCALE is an upper bound

of Dif[(A,D), (B,E)] = sigma_min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'T', SCALE is not touched.

- **DIF (output)**
  On exit SCALE is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. SCALE is an upper bound of Dif[(A,D), (B,E)] = sigma_min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'T', SCALE is not touched.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. If IJOB = 1 or 2 and TRANS = 'N', LWORK >= 2*M*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(M+N+2)`
- **INFO (output)**

```
=0: successful exit

<0: If INFO  = -i, the i-th argument had an illegal value.

>0: (A, D) and (B, E) have common or close eigenvalues.
```

FURTHER DETAILS

Based on contributions by


```
Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.
```


[1] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

[2] B. Kagstrom, A Perturbation Analysis of the Generalized Sylvester Equation (AR - LB, DR - LE) = (C, F), SIAM J. Matrix Anal. Appl., 15(4):1045-1060, 1994

[3] B. Kagstrom and L. Westin, Generalized Schur Methods with Condition Estimators for Solving the Generalized Sylvester Equation, IEEE Transactions on Automatic Control, Vol. 34, No. 7, July 1989, pp 745-751.

NAME

dtpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
  SUBROUTINE DTPCON( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2, INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  INTEGER N, INFO
  INTEGER WORK2(*)
  DOUBLE PRECISION RCOND
  DOUBLE PRECISION A(*), WORK(*)

  SUBROUTINE DTPCON_64( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2,
*       INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  INTEGER*8 N, INFO
  INTEGER*8 WORK2(*)
  DOUBLE PRECISION RCOND
  DOUBLE PRECISION A(*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE TPCON( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
*       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  INTEGER :: N, INFO
  INTEGER, DIMENSION(:) :: WORK2
  REAL(8) :: RCOND
  REAL(8), DIMENSION(:) :: A, WORK

  SUBROUTINE TPCON_64( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
*       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  INTEGER(8) :: N, INFO
  INTEGER(8), DIMENSION(:) :: WORK2
  REAL(8) :: RCOND
  REAL(8), DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void dtpcon(char norm, char uplo, char diag, int n, double *a, double *rcond, int *info);

void dtpcon_64(char norm, char uplo, char diag, long n, double *a, double *rcond, long *info);

PURPOSE

dtpcon estimates the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

dtpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE DTPMV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, INCY
DOUBLE PRECISION A(*), Y(*)

SUBROUTINE DTPMV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, INCY
DOUBLE PRECISION A(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TPMV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, INCY
REAL(8), DIMENSION(:) :: A, Y

SUBROUTINE TPMV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, INCY
REAL(8), DIMENSION(:) :: A, Y
```

C INTERFACE

#include <sunperf.h>

void dtpmv(char uplo, char transa, char diag, int n, double *a, double *y, int incy);

void dtpmv_64(char uplo, char transa, char diag, long n, double *a, double *y, long incy);

PURPOSE

dtpmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE DTPRFS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DTPRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TPRFS( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB], X,
*        [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: A, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X

 SUBROUTINE TPRFS_64( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: A, FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void dtprfs(char uplo, char transa, char diag, int n, int nrhs, double *a, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dtprfs_64(char uplo, char transa, char diag, long n, long nrhs, double *a, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dtprfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix.

The solution matrix X must be computed by STPTRS or some other means before entering this routine. STPRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <= i <= j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <= i <= n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dtpsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE DTPSV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, INCY
DOUBLE PRECISION A(*), Y(*)

SUBROUTINE DTPSV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, INCY
DOUBLE PRECISION A(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TPSV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, INCY
REAL(8), DIMENSION(:) :: A, Y

SUBROUTINE TPSV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, INCY
REAL(8), DIMENSION(:) :: A, Y
```

C INTERFACE

#include <sunperf.h>

void dtpsv(char uplo, char transa, char diag, int n, double *a, double *y, int incy);

void dtpsv_64(char uplo, char transa, char diag, long n, double *a, double *y, long incy);

PURPOSE

dtpsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtptri - compute the inverse of a real upper or lower triangular matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE DTPTRI( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, INFO
DOUBLE PRECISION A(*)

SUBROUTINE DTPTRI_64( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, INFO
DOUBLE PRECISION A(*)
```

F95 INTERFACE

```
SUBROUTINE TPTRI( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: A

SUBROUTINE TPTRI_64( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtptri(char uplo, char diag, int n, double *a, int *info);

void dtptri_64(char uplo, char diag, long n, double *a, long *info);

PURPOSE

dtptri computes the inverse of a real upper or lower triangular matrix A stored in packed format.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangular matrix A, stored columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = A(i,j) for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*((2*n-j)/2) = A(i,j) for j < =i < =n. See below for further details. On exit, the (triangular) inverse of the original matrix, in the same packed storage format.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

FURTHER DETAILS

A triangular matrix A can be transferred to packed storage using one of the following program segments:

UPLO = 'U': UPLO = 'L':

```
JC  = 1                          JC  = 1

DO 2 J  = 1, N                   DO 2 J  = 1, N
```

```
     DO 1 I  = 1, J
        A(JC+I-1)  = A(I,J)
1    CONTINUE
     JC  = JC + J
2 CONTINUE
```

```
      DO 1 I  = J, N
        A(JC+I-J)  = A(I,J)
1    CONTINUE
      JC  = JC + N - J + 1
2 CONTINUE
```

NAME

dtptrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
SUBROUTINE DTPTRS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)

SUBROUTINE DTPTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE TPTRS( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B

SUBROUTINE TPTRS_64( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: A
REAL(8), DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void dtptrs(char uplo, char transa, char diag, int n, int nrhs, double *a, double *b, int ldb, int *info);

void dtptrs_64(char uplo, char transa, char diag, long n, long nrhs, double *a, double *b, long ldb, long *info);

PURPOSE

dtptrs solves a triangular system of the form

where A is a triangular matrix of order N stored in packed format, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B  (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

 < 0: if INFO = -i, the i-th argument had an illegal value

 > 0: if INFO = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.

NAME

dtrans - transpose and scale source matrix

SYNOPSIS

```
SUBROUTINE DTRANS( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
INTEGER M, N
DOUBLE PRECISION SCALE
DOUBLE PRECISION SOURCE(*), DEST(*)

SUBROUTINE DTRANS_64( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
INTEGER*8 M, N
DOUBLE PRECISION SCALE
DOUBLE PRECISION SOURCE(*), DEST(*)
```

F95 INTERFACE

```
SUBROUTINE TRANS( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
INTEGER :: M, N
REAL(8) :: SCALE
REAL(8), DIMENSION(:) :: SOURCE, DEST

SUBROUTINE TRANS_64( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
INTEGER(8) :: M, N
REAL(8) :: SCALE
REAL(8), DIMENSION(:) :: SOURCE, DEST
```

C INTERFACE

#include <sunperf.h>

void dtrans(char place, double scale, double *source, int m, int n, double *dest);

void dtrans_64(char place, double scale, double *source, long m, long n, double *dest);

PURPOSE

dtrans scales and transposes the source matrix. The N2 x N1 result is written into SOURCE when PLACE = 'I' or 'i', and DEST when PLACE = 'O' or 'o'.

```
PLACE = 'I' or 'i': SOURCE = SCALE * SOURCE'

PLACE = 'O' or 'o': DEST = SCALE * SOURCE'
```

ARGUMENTS

- **PLACE (input)**
  Type of transpose. 'I' or 'i' for in-place, 'O' or 'o' for out-of-place. 'I' is default.
- **SCALE (input)**
  Scale factor on the SOURCE matrix.
- **SOURCE (input/output)**
  (M, N) on input. Array of (N, M) on output if in-place transpose.
- **M (input)**
  Number of rows in the SOURCE matrix on input.
- **N (input)**
  Number of columns in the SOURCE matrix on input.
- **DEST (output)**
  Scaled and transposed SOURCE matrix if out-of-place transpose. Not referenced if in-place transpose.

NAME

dtrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE DTRCON( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER N, LDA, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)

 SUBROUTINE DTRCON_64( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER*8 N, LDA, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRCON( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: A

 SUBROUTINE TRCON_64( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8) :: RCOND
```

```
REAL(8), DIMENSION(:) :: WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtrcon(char norm, char uplo, char diag, int n, double *a, int lda, double *rcond, int *info);

void dtrcon_64(char norm, char uplo, char diag, long n, double *a, long lda, double *rcond, long *info);

PURPOSE

dtrcon estimates the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dtrevc - compute some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T

SYNOPSIS

```
 SUBROUTINE DTREVC( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE DTREVC_64( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREVC( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*        VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: T, VL, VR

 SUBROUTINE TREVC_64( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*        VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: T, VL, VR
```

C INTERFACE

#include <sunperf.h>

void dtrevc(char side, char howmny, logical *select, int n, double *t, int ldt, double *vl, int ldvl, double *vr, int ldvr, int mm, int *m, int *info);

void dtrevc_64(char side, char howmny, logical *select, long n, double *t, long ldt, double *vl, long ldvl, double *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

dtrevc computes some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:

```
T*x = w*x,     y'*T = w*y'
```

where y' denotes the conjugate transpose of the vector y.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of T, or the products Q*X and/or Q*Y, where Q is an input orthogonal

matrix. If T was obtained from the real-Schur factorization of an original matrix A = Q*T*Q', then Q*X and Q*Y are the matrices of right or left eigenvectors of A.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign. Corresponding to each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R':  compute right eigenvectors only;

  = 'L':  compute left eigenvectors only;

  = 'B':  compute both right and left eigenvectors.
  ```

- **HOWMNY (input)**

  ```
  = 'A':  compute all right and/or left eigenvectors;

  = 'B':  compute all right and/or left eigenvectors,
and backtransform them using the input matrices
supplied in VR and/or VL;
 = 'S':  compute selected right and/or left eigenvectors,
  ```

specified by the logical array SELECT.

- **SELECT (input/output)**
  If HOWMNY = 'S', SELECT specifies the eigenvectors to be computed. If HOWMNY = 'A' or 'B', SELECT is not referenced. To select the real eigenvector corresponding to a real eigenvalue w(j), SELECT(j) must be set to .TRUE.. To select the complex eigenvector corresponding to a complex conjugate pair w(j) and w(j+1), either SELECT(j) or SELECT(j+1) must be set to .TRUE.; then on exit SELECT(j) is .TRUE. and SELECT(j+1) is .FALSE..
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  The upper quasi-triangular matrix T in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the orthogonal matrix Q of Schur vectors returned by SHSEQR). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of T; VL has the same quasi-lower triangular form as T'. If T(i,i) is a real eigenvalue, then the i-th column VL(i) of VL is its corresponding eigenvector. If T(i:i+1,i:i+1) is a 2-by-2 block whose eigenvalues are complex-conjugate eigenvalues of T, then VL(i)+sqrt(-1)*VL(i+1) is the complex eigenvector corresponding to the eigenvalue with positive real part. if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of T specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= max(1,N) if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the orthogonal matrix Q of Schur vectors returned by SHSEQR). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of T; VR has the same quasi-upper triangular form as T. If T(i,i) is a real eigenvalue, then the i-th column VR(i) of VR is its corresponding eigenvector. If T(i:i+1,i:i+1) is a 2-by-2 block whose eigenvalues are complex-conjugate eigenvalues of T, then VR(i)+sqrt(-1)*VR(i+1) is the complex eigenvector corresponding to the eigenvalue with positive real part. if HOWMNY = 'B', the matrix Q*X; if HOWMNY = 'S', the right eigenvectors of T specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= max(1,N) if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  dimension(3*N)
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

FURTHER DETAILS

The algorithm used in this program is basically backward (forward) substitution, with scaling to make the the code robust against possible overflow.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x| + |y|.

NAME

dtrexc - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST

SYNOPSIS

```
 SUBROUTINE DTREXC( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, WORK, INFO)
 CHARACTER * 1 COMPQ
 INTEGER N, LDT, LDQ, IFST, ILST, INFO
 DOUBLE PRECISION T(LDT,*), Q(LDQ,*), WORK(*)

 SUBROUTINE DTREXC_64( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, WORK,
*      INFO)
 CHARACTER * 1 COMPQ
 INTEGER*8 N, LDT, LDQ, IFST, ILST, INFO
 DOUBLE PRECISION T(LDT,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREXC( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: COMPQ
 INTEGER :: N, LDT, LDQ, IFST, ILST, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: T, Q

 SUBROUTINE TREXC_64( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: COMPQ
 INTEGER(8) :: N, LDT, LDQ, IFST, ILST, INFO
 REAL(8), DIMENSION(:) :: WORK
 REAL(8), DIMENSION(:,:) :: T, Q
```

C INTERFACE

#include <sunperf.h>

void dtrexc(char compq, int n, double *t, int ldt, double *q, int ldq, int *ifst, int *ilst, int *info);

void dtrexc_64(char compq, long n, double *t, long ldt, double *q, long ldq, long *ifst, long *ilst, long *info);

PURPOSE

dtrexc reorders the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST.

The real Schur form T is reordered by an orthogonal similarity transformation Z**T*T*Z, and optionally the matrix Q of Schur vectors is updated by postmultiplying it with Z.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **COMPQ (input)**

  ```
  = 'V':  update the matrix Q of Schur vectors;

  = 'N':  do not update Q.
  ```

- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input/output)**
  On entry, the upper quasi-triangular matrix T, in Schur Schur canonical form. On exit, the reordered upper quasi-triangular matrix, again in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the orthogonal transformation matrix Z which reorders T. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of T. The block with row index IFST is moved to row ILST, by a sequence of transpositions between adjacent blocks. On exit, if IFST pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; ILST always points to the first row of the block in its final position (which may differ from its input value by +1 or -1). 1 < = IFST < = N; 1 < = ILST < = N.
- **ILST (input/output)**
  See the description of IFST.
- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 = 1:  two adjacent blocks were too close to swap (the problem
is very ill-conditioned); T may have been partially
reordered, and ILST points to the first row of the
current position of the block being moved.
```

NAME

dtrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A)

SYNOPSIS

```
 SUBROUTINE DTRMM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER M, N, LDA, LDB
 DOUBLE PRECISION ALPHA
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DTRMM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER*8 M, N, LDA, LDB
 DOUBLE PRECISION ALPHA
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRMM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER :: M, N, LDA, LDB
 REAL(8) :: ALPHA
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE TRMM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER(8) :: M, N, LDA, LDB
 REAL(8) :: ALPHA
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dtrmm(char side, char uplo, char transa, char diag, int m, int n, double alpha, double *a, int lda, double *b, int ldb);

void dtrmm_64(char side, char uplo, char transa, char diag, long m, long n, double alpha, double *a, long lda, double *b, long ldb);

PURPOSE

dtrmm performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'.
```

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) multiplies B from the left or right as follows:

  SIDE = 'L' or 'l' B : = alpha*op(A)*B.

  SIDE = 'R' or 'r' B : = alpha*B*op(A).

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = A'.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be one. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), when SIDE = 'R' or 'r' then LDA > = max(1, n). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading m by n part of the array B must contain the matrix B, and on exit is overwritten by the transformed matrix.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.

NAME

dtrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE DTRMV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)

SUBROUTINE DTRMV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TRMV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TRMV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtrmv(char uplo, char transa, char diag, int n, double *a, int lda, double *y, int incy);

void dtrmv_64(char uplo, char transa, char diag, long n, double *a, long lda, double *y, long incy);

PURPOSE

dtrmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE DTRRFS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB, X,
*      LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDA, LDB, LDX, INFO
 INTEGER WORK2(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE DTRRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDA, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRRFS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDA, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, B, X

 SUBROUTINE TRRFS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDA, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK
 REAL(8), DIMENSION(:,:) :: A, B, X
```

C INTERFACE

#include <sunperf.h>

void dtrrfs(char uplo, char transa, char diag, int n, int nrhs, double *a, int lda, double *b, int ldb, double *x, int ldx, double *ferr, double *berr, int *info);

void dtrrfs_64(char uplo, char transa, char diag, long n, long nrhs, double *a, long lda, double *b, long ldb, double *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

dtrrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix.

The solution matrix X must be computed by STRTRS or some other means before entering this routine. STRRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is

not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

dtrsen - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T,

---

SYNOPSIS

```
 SUBROUTINE DTRSEN( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, WR, WI, M,
*       S, SEP, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 INTEGER N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION S, SEP
 DOUBLE PRECISION T(LDT,*), Q(LDQ,*), WR(*), WI(*), WORK(*)

 SUBROUTINE DTRSEN_64( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, WR, WI,
*       M, S, SEP, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 INTEGER*8 N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION S, SEP
 DOUBLE PRECISION T(LDT,*), Q(LDQ,*), WR(*), WI(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRSEN( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], WR,
*        WI, M, S, SEP, WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 INTEGER :: N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8) :: S, SEP
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: T, Q
```

```
 SUBROUTINE TRSEN_64( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ],
*       WR, WI, M, S, SEP, WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 INTEGER(8) :: N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8) :: S, SEP
 REAL(8), DIMENSION(:) :: WR, WI, WORK
 REAL(8), DIMENSION(:,:) :: T, Q
```

C INTERFACE

#include <sunperf.h>

void dtrsen(char job, char compq, logical *select, int n, double *t, int ldt, double *q, int ldq, double *wr, double *wi, int *m, double *s, double *sep, double *work, int lwork, int *info);

void dtrsen_64(char job, char compq, logical *select, long n, double *t, long ldt, double *q, long ldq, double *wr, double *wi, long *m, double *s, double *sep, double *work, long lwork, long *info);

PURPOSE

dtrsen reorders the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elemnts equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (S) or the invariant subspace (SEP):

  ```
  = 'N': none;

  = 'E': for eigenvalues only (S);

  = 'V': for invariant subspace only (SEP);

  = 'B': for both eigenvalues and invariant subspace (S and
  SEP).
  ```

- **COMPQ (input)**

  ```
  = 'V': update the matrix Q of Schur vectors;
  ```

= 'N': do not update Q.

- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select a real eigenvalue w(j), SELECT(j) must be set to .TRUE.. To select a complex conjugate pair of eigenvalues w(j) and w(j+1), corresponding to a 2-by-2 diagonal block, either SELECT(j) or SELECT(j+1) or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  On entry, the upper quasi-triangular matrix T, in Schur canonical form. On exit, T is overwritten by the reordered matrix T, again in Schur canonical form, with the selected eigenvalues in the leading diagonal blocks.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the orthogonal transformation matrix which reorders T; the leading M columns of Q form an orthonormal basis for the specified invariant subspace. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; and if COMPQ = 'V', LDQ >= N.
- **WR (output)**
  The real and imaginary parts, respectively, of the reordered eigenvalues of T. The eigenvalues are stored in the same order as on the diagonal of T, with WR(i) = T(i,i) and, if T(i:i+1,i:i+1) is a 2-by-2 diagonal block, WI(i) > 0 and WI(i+1) = -WI(i). Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
- **WI (output)**
  See the description of WR.
- **M (output)**
  The dimension of the specified invariant subspace. 0 <= M <= N.
- **S (output)**
  If JOB = 'E' or 'B', S is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. S cannot underestimate the true reciprocal condition number by more than a factor of sqrt(N). If M = 0 or N, S = 1. If JOB = 'N' or 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', SEP is the estimated reciprocal condition number of the specified invariant subspace. If M = 0 or N, SEP = norm(T). If JOB = 'N' or 'E', SEP is not referenced.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOB = 'N', LWORK >= max(1,N); if JOB = 'E', LWORK >= M*(N-M); if JOB = 'V' or 'B', LWORK >= 2*M*(N-M).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If JOB = 'N' or 'E', IWORK is not referenced.
- **LIWORK (input)**
  The dimension of the array IWORK. If JOB = 'N' or 'E', LIWORK >= 1; if JOB = 'V' or 'B', LIWORK >= M*(N-M).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1: reordering of T failed because some eigenvalues are too
close to separate (the problem is very ill-conditioned);
T may have been partially reordered, and WR and WI contain
the eigenvalues in the same order as in T; S and SEP
(if requested) are set to zero.
```

FURTHER DETAILS

STRSEN first collects the selected eigenvalues by computing an orthogonal transformation Z to move them to the top left corner of T. In other words, the selected eigenvalues are the eigenvalues of T11 in:

```
            Z'*T*Z  = ( T11 T12 ) n1

                      (  0  T22 ) n2

                         n1  n2
```

where N = n1+n2 and Z' means the transpose of Z. The first n1 columns of Z span the specified invariant subspace of T.

If T has been obtained from the real Schur factorization of a matrix A = Q*T*Q', then the reordered real Schur factorization of A is given by A = (Q*Z)*(Z'*T*Z)*(Q*Z)', and the first n1 columns of Q*Z span the corresponding invariant subspace of A.

The reciprocal condition number of the average of the eigenvalues of T11 may be returned in S. S lies between 0 (very badly conditioned) and 1 (very well conditioned). It is computed as follows. First we compute R so that

```
                      P  = ( I  R ) n1

                           ( 0  0 ) n2

                             n1 n2
```

is the projector on the invariant subspace associated with T11. R is the solution of the Sylvester equation:

```
                     T11*R - R*T22  = T12.
```

Let F-norm(M) denote the Frobenius-norm of M and 2-norm(M) denote the two-norm of M. Then S is computed as the lower bound

```
                   (1 + F-norm(R)**2)**(-1/2)
```

on the reciprocal of 2-norm(P), the true reciprocal condition number. S cannot underestimate 1 / 2-norm(P) by more than a factor of sqrt(N).

An approximate error bound for the computed average of the eigenvalues of T11 is

```
                      EPS * norm(T) / S
```

where EPS is the machine precision.

The reciprocal condition number of the right invariant subspace spanned by the first n1 columns of Z (or of Q*Z) is returned in SEP. SEP is defined as the separation of T11 and T22:

```
sep( T11, T22 )  = sigma-min( C )
```

where sigma-min(C) is the smallest singular value of the

n1*n2-by-n1*n2 matrix

```
C   = kprod( I(n2), T11 ) - kprod( transpose(T22), I(n1) )
```

`I(m)` is an m by m identity matrix, and kprod denotes the Kronecker product. We estimate sigma-min(C) by the reciprocal of an estimate of the 1-norm of inverse(C). The true reciprocal 1-norm of `inverse(C)` cannot differ from sigma-min(C) by more than a factor of sqrt(n1*n2).

When SEP is small, small changes in T can cause large changes in the invariant subspace. An approximate bound on the maximum angular error in the computed right invariant subspace is

```
EPS * norm(T) / SEP
```

NAME

dtrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

SYNOPSIS

```
 SUBROUTINE DTRSM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER M, N, LDA, LDB
 DOUBLE PRECISION ALPHA
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DTRSM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER*8 M, N, LDA, LDB
 DOUBLE PRECISION ALPHA
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER :: M, N, LDA, LDB
 REAL(8) :: ALPHA
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE TRSM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER(8) :: M, N, LDA, LDB
 REAL(8) :: ALPHA
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dtrsm(char side, char uplo, char transa, char diag, int m, int n, double alpha, double *a, int lda, double *b, int ldb);

void dtrsm_64(char side, char uplo, char transa, char diag, long m, long n, double alpha, double *a, long lda, double *b, long ldb);

PURPOSE

dtrsm solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'.
```

The matrix X is overwritten on B.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) appears on the left or right of X as follows:

  SIDE = 'L' or 'l' op(A)*X = alpha*B.

  SIDE = 'R' or 'r' X*op(A) = alpha*B.

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = A'.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be one. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), when SIDE = 'R' or 'r' then LDA > = max(1, n). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading m by n part of the array B must contain the right-hand side matrix B, and on exit is overwritten by the solution matrix X.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.

NAME

dtrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal)

SYNOPSIS

```
 SUBROUTINE DTRSNA( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 INTEGER N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER WORK1(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION T(LDT,*), VL(LDVL,*), VR(LDVR,*), S(*), SEP(*), WORK(LDWORK,*)

 SUBROUTINE DTRSNA_64( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER*8 WORK1(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION T(LDT,*), VL(LDVL,*), VR(LDVR,*), S(*), SEP(*), WORK(LDWORK,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSNA( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER, DIMENSION(:) :: WORK1
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, SEP
 REAL(8), DIMENSION(:,:) :: T, VL, VR, WORK

 SUBROUTINE TRSNA_64( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
```

```
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: WORK1
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8), DIMENSION(:) :: S, SEP
REAL(8), DIMENSION(:,:) :: T, VL, VR, WORK
```

C INTERFACE

#include <sunperf.h>

void dtrsna(char job, char howmny, logical *select, int n, double *t, int ldt, double *vl, int ldvl, double *vr, int ldvr, double *s, double *sep, int mm, int *m, int ldwork, int *info);

void dtrsna_64(char job, char howmny, logical *select, long n, double *t, long ldt, double *vl, long ldvl, double *vr, long ldvr, double *s, double *sep, long mm, long *m, long ldwork, long *info);

PURPOSE

dtrsna estimates reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal).

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (SEP):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (SEP);

  = 'B': for both eigenvalues and eigenvectors (S and SEP).
  ```
- **HOWMNY (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNY = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the eigenpair corresponding to a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select condition numbers corresponding to a complex conjugate pair of eigenvalues `w(j)` and w(j+1), either `SELECT(j)` or `SELECT(j+1)` or both, must be set to .TRUE.. If HOWMNY = 'A', SELECT is not referenced.
- **N (input)**
  The order of the matrix T. N >= 0.

- **T (input)**
  The upper quasi-triangular matrix T, in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of T (or of any Q*T*Q**T with Q orthogonal), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by SHSEIN or STREVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and if JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of T (or of any Q*T*Q**T with Q orthogonal), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by SHSEIN or STREVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; and if JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of S are set to the same value. Thus S(j), SEP(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of SEP are set to the same value. If the eigenvalues cannot be reordered to compute SEP(j), `SEP(j)` is set to 0; this can only occur when the true value would be very small anyway. If JOB = 'E', SEP is not referenced.
- **MM (input)**
  The number of elements in the arrays S (if JOB = 'E' or 'B') and/or SEP (if JOB = 'V' or 'B'). MM > = M.
- **M (output)**
  The number of elements of the arrays S and/or SEP actually used to store the estimated condition numbers. If HOWMNY = 'A', M is set to N.
- **WORK (workspace)**
  `dimension(LDWORK,N+1)` If JOB = 'E', WORK is not referenced.
- **LDWORK (input)**
  The leading dimension of the array WORK. LDWORK > = 1; and if JOB = 'V' or 'B', LDWORK > = N.
- **WORK1 (workspace)**
  `dimension(N)` If JOB = 'E', WORK1 is not referenced.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of an eigenvalue lambda is defined as

```
S(lambda)  = |v'*u| / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of T corresponding to lambda; v' denotes the conjugate-transpose of v, and `norm(u)` denotes the Euclidean norm. These reciprocal condition numbers always lie between zero (very badly conditioned) and one (very well conditioned). If n = 1, `S(lambda)` is defined to be 1.

An approximate error bound for a computed eigenvalue `W(i)` is given by

```
EPS * norm(T) / S(i)
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u corresponding to lambda is defined as follows. Suppose

```
T  = ( lambda  c  )

     (   0    T22 )
```

Then the reciprocal condition number is

```
SEP( lambda, T22 )  = sigma-min( T22 - lambda*I )
```

where sigma-min denotes the smallest singular value. We approximate the smallest singular value by the reciprocal of an estimate of the one-norm of the inverse of T22 - lambda*I. If n = 1, `SEP(1)` is defined to be abs(T(1,1)).

An approximate error bound for a computed right eigenvector `VR(i)` is given by

```
EPS * norm(T) / SEP(i)
```

NAME

dtrsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE DTRSV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)

SUBROUTINE DTRSV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, LDA, INCY
DOUBLE PRECISION A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TRSV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TRSV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, LDA, INCY
REAL(8), DIMENSION(:) :: Y
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtrsv(char uplo, char transa, char diag, int n, double *a, int lda, double *y, int incy);

void dtrsv_64(char uplo, char transa, char diag, long n, double *a, long lda, double *y, long incy);

PURPOSE

dtrsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**

On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

dtrsyl - solve the real Sylvester matrix equation

SYNOPSIS

```
 SUBROUTINE DTRSYL( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C, LDC,
*       SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 INTEGER ISGN, M, N, LDA, LDB, LDC, INFO
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE DTRSYL_64( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C,
*       LDC, SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 INTEGER*8 ISGN, M, N, LDA, LDB, LDC, INFO
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSYL( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 INTEGER :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:,:) :: A, B, C

 SUBROUTINE TRSYL_64( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 INTEGER(8) :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void dtrsyl(char trana, char tranb, int isgn, int m, int n, double *a, int lda, double *b, int ldb, double *c, int ldc, double *scale, int *info);

void dtrsyl_64(char trana, char tranb, long isgn, long m, long n, double *a, long lda, double *b, long ldb, double *c, long ldc, double *scale, long *info);

PURPOSE

dtrsyl solves the real Sylvester matrix equation:

```
op(A)*X + X*op(B) = scale*C or

op(A)*X - X*op(B) = scale*C,
```

where `op(A)` = A or A**T, and A and B are both upper quasi- triangular. A is M-by-M and B is N-by-N; the right hand side C and the solution X are M-by-N; and scale is an output scale factor, set <= 1 to avoid overflow in X.

A and B must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **TRANA (input)**
  Specifies the option op(A):

  ```
  = 'N': op(A)  = A    (No transpose)

  = 'T': op(A)  = A**T (Transpose)

  = 'C': op(A)  = A**H (Conjugate transpose  = Transpose)
  ```
- **TRANB (input)**
  Specifies the option op(B):

  ```
  = 'N': op(B)  = B    (No transpose)

  = 'T': op(B)  = B**T (Transpose)

  = 'C': op(B)  = B**H (Conjugate transpose  = Transpose)
  ```
- **ISGN (input)**
  Specifies the sign in the equation:

  ```
  = +1: solve op(A)*X + X*op(B)  = scale*C
  ```

= -1: solve op(A)*X - X*op(B) = scale*C

- **M (input)**
  The order of the matrix A, and the number of rows in the matrices X and C. M >= 0.
- **N (input)**
  The order of the matrix B, and the number of columns in the matrices X and C. N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A, in Schur canonical form.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input)**
  The upper quasi-triangular matrix B, in Schur canonical form.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **C (input/output)**
  On entry, the M-by-N right hand side matrix C. On exit, C is overwritten by the solution matrix X.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)`
- **SCALE (output)**
  The scale factor, scale, set <= 1 to avoid overflow in X.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

= 1: A and B have common or very close eigenvalues; perturbed
values were used to solve the equation (but the matrices
A and B are unchanged).

NAME

dtrti2 - compute the inverse of a real upper or lower triangular matrix

---

SYNOPSIS

```
SUBROUTINE DTRTI2( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DTRTI2_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE TRTI2( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TRTI2_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtrti2(char uplo, char diag, int n, double *a, int lda, int *info);

void dtrti2_64(char uplo, char diag, long n, double *a, long lda, long *info);

PURPOSE

dtrti2 computes the inverse of a real upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **DIAG (input)**
  Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular

  ```
  = 'U':  Unit triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading n by n upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

  On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

dtrtri - compute the inverse of a real upper or lower triangular matrix A

SYNOPSIS

```
SUBROUTINE DTRTRI( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, LDA, INFO
DOUBLE PRECISION A(LDA,*)

SUBROUTINE DTRTRI_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE TRTRI( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TRTRI_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, LDA, INFO
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtrtri(char uplo, char diag, int n, double *a, int lda, int *info);

void dtrtri_64(char uplo, char diag, long n, double *a, long lda, long *info);

PURPOSE

dtrtri computes the inverse of a real upper or lower triangular matrix A.

This is the Level 3 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1. On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value

> 0: if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

NAME

dtrtrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
 SUBROUTINE DTRTRS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDA, LDB, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*)

 SUBROUTINE DTRTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDA, LDB, INFO
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRTRS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDA, LDB, INFO
 REAL(8), DIMENSION(:,:) :: A, B

 SUBROUTINE TRTRS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDA, LDB, INFO
 REAL(8), DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void dtrtrs(char uplo, char transa, char diag, int n, int nrhs, double *a, int lda, double *b, int ldb, int *info);

void dtrtrs_64(char uplo, char transa, char diag, long n, long nrhs, double *a, long lda, double *b, long ldb, long *info);

PURPOSE

dtrtrs solves a triangular system of the form

where A is a triangular matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**

On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the solutions
X have not been computed.
```

NAME

dtzrqf - routine is deprecated and has been replaced by routine STZRZF

SYNOPSIS

```
SUBROUTINE DTZRQF( M, N, A, LDA, TAU, INFO)
INTEGER M, N, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*)

SUBROUTINE DTZRQF_64( M, N, A, LDA, TAU, INFO)
INTEGER*8 M, N, LDA, INFO
DOUBLE PRECISION A(LDA,*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE TZRQF( [M], [N], A, [LDA], TAU, [INFO])
INTEGER :: M, N, LDA, INFO
REAL(8), DIMENSION(:) :: TAU
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TZRQF_64( [M], [N], A, [LDA], TAU, [INFO])
INTEGER(8) :: M, N, LDA, INFO
REAL(8), DIMENSION(:) :: TAU
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtzrqf(int m, int n, double *a, int lda, double *tau, int *info);

void dtzrqf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dtzrqf routine is deprecated and has been replaced by routine STZRZF.

STZRQF reduces the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N orthogonal matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= M.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the orthogonal matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
Z( k )  = ( I     0   ),

          ( 0  T( k ) )
```

where

```
T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                             (   0    )
```

```
                                        ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

dtzrzf - reduce the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations

---

SYNOPSIS

```
SUBROUTINE DTZRZF( M, N, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER M, N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)

SUBROUTINE DTZRZF_64( M, N, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER*8 M, N, LDA, LWORK, INFO
DOUBLE PRECISION A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE TZRZF( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
INTEGER :: M, N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A

SUBROUTINE TZRZF_64( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
INTEGER(8) :: M, N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: TAU, WORK
REAL(8), DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void dtzrzf(int m, int n, double *a, int lda, double *tau, int *info);

void dtzrzf_64(long m, long n, double *a, long lda, double *tau, long *info);

PURPOSE

dtzrzf reduces the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N orthogonal matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the orthogonal matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

```
  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
   Z( k )  = ( I     0   ),

             ( 0  T( k ) )
```

where

```
   T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                                (   0    )
                                                ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

dwiener - perform Wiener deconvolution of two signals

SYNOPSIS

```
SUBROUTINE DWIENER( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER N_POINTS, ISW, IERR
DOUBLE PRECISION ACOR(*), XCOR(*), FLTR(*), EROP(*)

SUBROUTINE DWIENER_64( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER*8 N_POINTS, ISW, IERR
DOUBLE PRECISION ACOR(*), XCOR(*), FLTR(*), EROP(*)
```

F95 INTERFACE

```
SUBROUTINE WIENER( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER :: N_POINTS, ISW, IERR
REAL(8), DIMENSION(:) :: ACOR, XCOR, FLTR, EROP

SUBROUTINE WIENER_64( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER(8) :: N_POINTS, ISW, IERR
REAL(8), DIMENSION(:) :: ACOR, XCOR, FLTR, EROP
```

C INTERFACE

#include <sunperf.h>

void dwiener(int n_points, double *acor, double *xcor, double *fltr, double *erop, int *isw, int *ierr);

void dwiener_64(long n_points, double *acor, double *xcor, double *fltr, double *erop, long *isw, long *ierr);

PURPOSE

dwiener performs Wiener deconvolution of two signals.

ARGUMENTS

- **N_POINTS (input)**
  On entry, the number of points in the input correlations. Unchanged on exit.
- **ACOR (input)**
  On entry, autocorrelation coefficients. Unchanged on exit.
- **XCOR (input)**
  On entry, cross-correlation coefficients. Unchanged on exit.
- **FLTR (output)**
  On exit, filter coefficients. Unchanged on exit.
- **EROP (input)**
  On exit, the prediction error.
- **ISW (input/output)**
  On entry, if ISW .EQ. 0 then perform spiking deconvolution, otherwise perform general deconvolution. Unchanged on exit.
- **IERR (input/output)**
  On exit, the deconvolution was successful iff IERR .EQ. 0, otherwise there was an error.

NAME

dzasum - Return the sum of the absolute values of a vector x.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DZASUM( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER N, INCX

DOUBLE PRECISION FUNCTION DZASUM_64( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
REAL(8) FUNCTION ASUM( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER :: N, INCX

REAL(8) FUNCTION ASUM_64( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

double dzasum(int n, doublecomplex *x, int incx);

double dzasum_64(long n, doublecomplex *x, long incx);

PURPOSE

dzasum Return the sum of the absolute values of the elements of x where x is an n-vector. This is the sum of the absolute values of the real and complex elements and not the sum of the squares of the real and complex elements.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.

NAME

dznrm2 - Return the Euclidian norm of a vector.

SYNOPSIS

```
DOUBLE PRECISION FUNCTION DZNRM2( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER N, INCX

DOUBLE PRECISION FUNCTION DZNRM2_64( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
REAL(8) FUNCTION NRM2( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER :: N, INCX

REAL(8) FUNCTION NRM2_64( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

double dznrm2(int n, doublecomplex *x, int incx);

double dznrm2_64(long n, doublecomplex *x, long incx);

PURPOSE

dznrm2 Return the Euclidian norm of a vector x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NAME

ezfftb - computes a periodic sequence from its Fourier coefficients. EZFFTB is a simplified but slower version of RFFTB. =head1 SYNOPSIS

```
SUBROUTINE EZFFTB( N, R, AZERO, A, B, WSAVE)
INTEGER N
REAL AZERO
REAL R(*), A(*), B(*), WSAVE(*)

SUBROUTINE EZFFTB_64( N, R, AZERO, A, B, WSAVE)
INTEGER*8 N
REAL AZERO
REAL R(*), A(*), B(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE EZFFTB( N, R, AZERO, A, B, WSAVE)
INTEGER :: N
REAL :: AZERO
REAL, DIMENSION(:) :: R, A, B, WSAVE

SUBROUTINE EZFFTB_64( N, R, AZERO, A, B, WSAVE)
INTEGER(8) :: N
REAL :: AZERO
REAL, DIMENSION(:) :: R, A, B, WSAVE
```

C INTERFACE

#include <sunperf.h>

void ezfftb(int n, float *r, float azero, float *a, float *b, float *wsave);

void ezfftb_64(long n, float *r, float azero, float *a, float *b, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be synthesized. The method is most efficient when N is the product of small primes. N >= 0.
- **R (output)**
  On exit, the Fourier synthesis of the inputs.
- **AZERO (input)**
  On entry, the constant Fourier coefficient A0. Unchanged on exit.
- **A (input)**
  On entry, arrays that contain the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **B (input)**
  On entry, arrays that contain the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by EZFFTI.

NAME

ezfftf - computes the Fourier coefficients of a periodic sequence. EZFFTF is a simplified but slower version of RFFTF.
=head1 SYNOPSIS

```
SUBROUTINE EZFFTF( N, R, AZERO, A, B, WSAVE)
INTEGER N
REAL AZERO
REAL R(*), A(*), B(*), WSAVE(*)

SUBROUTINE EZFFTF_64( N, R, AZERO, A, B, WSAVE)
INTEGER*8 N
REAL AZERO
REAL R(*), A(*), B(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE EZFFTF( N, R, AZERO, A, B, WSAVE)
INTEGER :: N
REAL :: AZERO
REAL, DIMENSION(:) :: R, A, B, WSAVE

SUBROUTINE EZFFTF_64( N, R, AZERO, A, B, WSAVE)
INTEGER(8) :: N
REAL :: AZERO
REAL, DIMENSION(:) :: R, A, B, WSAVE
```

C INTERFACE

#include <sunperf.h>

void ezfftf(int n, float *r, float azero, float *a, float *b, float *wsave);

void ezfftf_64(long n, float *r, float azero, float *a, float *b, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is the product of small primes. N >= 0.
- **R (output)**
  A real array of length N containing the sequence to be transformed. On exit, R is unchanged.
- **AZERO (input)**
  On exit, the sum from i =1 to i =n of r(i)/n.
- **A (input)**
  On entry, arrays that contain the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **B (input)**
  On entry, arrays that contain the remaining Fourier coefficients. On exit, these arrays are unchanged.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least (3 * N + 15), initialized by EZFFTI.

NAME

ezffti - initializes the array WSAVE, which is used in both EZFFTF and EZFFTB. =head1 SYNOPSIS

```
SUBROUTINE EZFFTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE EZFFTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE EZFFTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE EZFFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void ezffti(int n, float *wsave);

void ezffti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array with a dimension of at least (3 * N + 15). The same work array can be used for both EZFFTF and EZFFTB as long as N remains unchanged. Different WSAVE arrays are required for different values of N. This initialization does not have to be repeated between calls to EZFFTF or EZFFTB as long as N and WSAVE remain unchanged, thus subsequent transforms can be obtained faster than the first.

NAME

icamax - return the index of the element with largest absolute value.

SYNOPSIS

```
INTEGER FUNCTION ICAMAX( N, X, INCX)
COMPLEX X(*)
INTEGER N, INCX

INTEGER*8 FUNCTION ICAMAX_64( N, X, INCX)
COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
INTEGER FUNCTION IAMAX( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER :: N, INCX

INTEGER(8) FUNCTION IAMAX_64( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

int icamax(int n, complex *x, int incx);

long icamax_64(long n, complex *x, long incx);

PURPOSE

icamax return the index of the element in x with largest absolute value where x is an n-vector and absolute value is defined as the sum of the absolute value of the real part and the absolute value of the imaginary part.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NOTES

If the vector contains all NaNs, the function returns 1. If the vector contains valid complex numbers and one or more NaNs, the routine returns the index of the element containing the largest absolute value.

NAME

idamax - return the index of the element with largest absolute value.

SYNOPSIS

```
INTEGER FUNCTION IDAMAX( N, X, INCX)
INTEGER N, INCX
DOUBLE PRECISION X(*)

INTEGER*8 FUNCTION IDAMAX_64( N, X, INCX)
INTEGER*8 N, INCX
DOUBLE PRECISION X(*)
```

F95 INTERFACE

```
INTEGER FUNCTION IAMAX( [N], X, [INCX])
INTEGER :: N, INCX
REAL(8), DIMENSION(:) :: X

INTEGER(8) FUNCTION IAMAX_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL(8), DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

int idamax(int n, double *x, int incx);

long idamax_64(long n, double *x, long incx);

PURPOSE

idamax return the index of the element in x with largest absolute value where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NOTES

If the vector contains all NaNs, the function returns 1. If the vector contains valid floating point numbers and one or more NaNs, the routine returns the index of the lement containing the largest absolute value.

NAME

The name of the calling subroutine, in either upper case or lower case. =head1 SYNOPSIS

```
INTEGER FUNCTION ILAENV( ISPEC, NAME, OPTS, N1, N2, N3, N4)
CHARACTER * 1 NAME(*), OPTS(*)
INTEGER ISPEC, N1, N2, N3, N4

INTEGER*8 FUNCTION ILAENV_64( ISPEC, NAME, OPTS, N1, N2, N3, N4)
CHARACTER * 1 NAME(*), OPTS(*)
INTEGER*8 ISPEC, N1, N2, N3, N4
```

F95 INTERFACE

```
INTEGER FUNCTION ILAENV( ISPEC, NAME, OPTS, N1, N2, N3, N4)
CHARACTER(LEN=1), DIMENSION(:) :: NAME, OPTS
INTEGER :: ISPEC, N1, N2, N3, N4

INTEGER(8) FUNCTION ILAENV_64( ISPEC, NAME, OPTS, N1, N2, N3, N4)
CHARACTER(LEN=1), DIMENSION(:) :: NAME, OPTS
INTEGER(8) :: ISPEC, N1, N2, N3, N4
```

C INTERFACE

#include <sunperf.h>

int ilaenv(int ispec, char *name, char *opts, int n1, int n2, int n3, int n4);

long ilaenv_64(long ispec, char *name, char *opts, long n1, long n2, long n3, long n4);

PURPOSE

ilaenv is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See ISPEC for a description of the parameters.

This version provides a set of parameters which should give good, but not optimal, performance on many of the currently available computers. Users are encouraged to modify this subroutine to set the tuning parameters for their particular machine using the option and problem size information in the arguments.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

ARGUMENTS

- **ISPEC (input)**
  Specifies the parameter to be returned as the value of ILAENV. = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance. = 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used. = 3: the crossover point (in a block routine, for N less than this value, an unblocked routine should be used) = 4: the number of shifts, used in the nonsymmetric eigenvalue routines = 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k by m, where k is given by `ILAENV(2,...)` and m by `ILAENV(5,...)` = 6: the crossover point for the SVD (when reducing an m by n matrix to bidiagonal form, if `max(m,n)/min(m,n)` exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form.) = 7: the number of processors

  ```
   = 8: the crossover point for the multishift QR and QZ methods
  for nonsymmetric eigenvalue problems.
   = 9: maximum size of the subproblems at the bottom of the
  computation tree in the divide-and-conquer algorithm
  (used by xGELSD and xGESDD)
   =10: ieee NaN arithmetic can be trusted not to trap

   =11: infinity arithmetic can be trusted not to trap
  ```

- **NAME (input)**
  The name of the calling subroutine, in either upper case or lower case.
- **OPTS (input)**
  The character options to the subroutine NAME, concatenated into a single character string. For example, UPLO = 'U', TRANS = 'T', and DIAG = 'N' for a triangular routine would be specified as OPTS = 'UTN'.
- **N1 (input)**

  ```
  INTEGER
  ```

- **N2 (input)**

  ```
  INTEGER
  ```

- **N3 (input)**

  ```
  INTEGER
  ```

- **N4 (input)**

  ```
  INTEGER
  ```

N1, N2, N3, N4 are problem dimensions for the subroutine NAME; these may not all be required.

```
> = 0: the value of the parameter specified by ISPEC

< 0:  if ILAENV  = -k, the k-th argument had an illegal value.

< 0:  if ILAENV  = -k, the k-th argument had an illegal value.
```

FURTHER DETAILS

The following conventions have been used when calling ILAENV from the LAPACK routines:

1) OPTS is a concatenation of all of the character options to subroutine NAME, in the same order that they appear in the argument list for NAME, even if they are not used in determining the value of the parameter specified by ISPEC.

2) The problem dimensions N1, N2, N3, N4 are specified in the order that they appear in the argument list for NAME. N1 is used first, N2 second, and so on, and unused problem dimensions are passed a value of -1.

3) The parameter value returned by ILAENV is checked for validity in the calling subroutine. For example, ILAENV is used to retrieve the optimal blocksize for STRTRI as follows:

```
NB  = ILAENV( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 )
IF( NB.LE.1 ) NB  = MAX( 1, N )
```

NAME

isamax - return the index of the element with largest absolute value.

SYNOPSIS

```
INTEGER FUNCTION ISAMAX( N, X, INCX)
INTEGER N, INCX
REAL X(*)

INTEGER*8 FUNCTION ISAMAX_64( N, X, INCX)
INTEGER*8 N, INCX
REAL X(*)
```

F95 INTERFACE

```
INTEGER FUNCTION IAMAX( [N], X, [INCX])
INTEGER :: N, INCX
REAL, DIMENSION(:) :: X

INTEGER(8) FUNCTION IAMAX_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

int isamax(int n, float *x, int incx);

long isamax_64(long n, float *x, long incx);

PURPOSE

isamax return the index of the element in x with largest absolute value where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NOTES

If the vector contains all NaNs, the function returns 1. If the vector contains valid floating point numbers and one or more NaNs, the routine returns the index of the element containing the largest absolute value.

NAME

izamax - return the index of the element with largest absolute value.

SYNOPSIS

```
INTEGER FUNCTION IZAMAX( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER N, INCX

INTEGER*8 FUNCTION IZAMAX_64( N, X, INCX)
DOUBLE COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
INTEGER FUNCTION IAMAX( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER :: N, INCX

INTEGER(8) FUNCTION IAMAX_64( [N], X, [INCX])
COMPLEX(8), DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

int izamax(int n, doublecomplex *x, int incx);

long izamax_64(long n, doublecomplex *x, long incx);

PURPOSE

izamax return the index of the element in x with largest absolute value where x is an n-vector and absolute value is defined as the sum of the absolute value of the real part and the absolute value of the imaginary part.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NOTES

If the vector contains all NaNs, the function returns 1. If the vector contains valid double complex numbers and one or more NaNs, the routine returns the index of the element containing the largest absolute value.

NAME

lsame - returns .TRUE. if CA is the same letter as CB regardless of case =head1 SYNOPSIS

```
LOGICAL FUNCTION LSAME( CA, CB)
CHARACTER * 1 CA, CB

LOGICAL*8 FUNCTION LSAME_64( CA, CB)
CHARACTER * 1 CA, CB
```

F95 INTERFACE

```
LOGICAL FUNCTION LSAME( CA, CB)
CHARACTER(LEN=1) :: CA, CB

LOGICAL(8) FUNCTION LSAME_64( CA, CB)
CHARACTER(LEN=1) :: CA, CB
```

C INTERFACE

#include <sunperf.h>

logical lsame(char ca, char cb);

logical lsame_64(char ca, char cb);

PURPOSE

lsame returns .TRUE. if CA is the same letter as CB regardless of case.

ARGUMENTS

- **CA (input)**
  On entry, CA is a single character to compare with CB. Unchanged on exit.
- **CB (input)**
  On entry, CB is a single character to compare with CA. Unchanged on exit.

NAME

rfft2b - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFT2F followed by a call of RFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
SUBROUTINE RFFT2B( PLACE, M, N, A, LDA, B, LDB, WORK, LWORK)
CHARACTER * 1 PLACE
INTEGER M, N, LDA, LDB, LWORK
REAL A(LDA,*), B(LDB,*), WORK(*)

SUBROUTINE RFFT2B_64( PLACE, M, N, A, LDA, B, LDB, WORK, LWORK)
CHARACTER * 1 PLACE
INTEGER*8 M, N, LDA, LDB, LWORK
REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2B( PLACE, [M], [N], A, [LDA], B, [LDB], WORK, LWORK)
CHARACTER(LEN=1) :: PLACE
INTEGER :: M, N, LDA, LDB, LWORK
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE FFT2B_64( PLACE, [M], [N], A, [LDA], B, [LDB], WORK,
*       LWORK)
CHARACTER(LEN=1) :: PLACE
INTEGER(8) :: M, N, LDA, LDB, LWORK
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void rfft2b(char place, int m, int n, float *a, int lda, float *b, int ldb, float *work, int lwork);

void rfft2b_64(char place, long m, long n, float *a, long lda, float *b, long ldb, float *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Character. If PLACE = 'I' or 'i' (for in-place) , the input and output data are stored in array A. If PLACE = 'O' or 'o' (for out-of-place), the input data is stored in array B while the output is stored in A.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  Real array of dimension (LDA,N). On entry, the two-dimensional array `A(LDA,N)` contains the input data to be transformed if an in-place transform is requested. Otherwise, it is not referenced. Upon exit, results are stored in A(1:M,1:N).
- **LDA (input)**
  Integer specifying the leading dimension of A. If an out-of-place transform is desired LDA > = M. Else if an in-place transform is desired LDA > = 2*(M/2+1).
- **B (input/output)**
  Real array of dimension (2*LDB, N). On entry, if an out-of-place transform is requested B contains the input data. Otherwise, B is not referenced. B is unchanged upon exit.
- **LDB (input)**
  Integer. If an out-of-place transform is desired, 2*LDB is the leading dimension of the array B which contains the data to be transformed and 2*LDB > = 2*(M/2+1). Otherwise it is not referenced.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by RFFT2I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*N + MAX(M, 2*N) + 30)

NAME

rfft2f - compute the Fourier coefficients of a periodic sequence. The RFFT operations are unnormalized, so a call of RFFT2F followed by a call of RFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
 SUBROUTINE RFFT2F( PLACE, FULL, M, N, A, LDA, B, LDB, WORK, LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER M, N, LDA, LDB, LWORK
 REAL A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE RFFT2F_64( PLACE, FULL, M, N, A, LDA, B, LDB, WORK,
*      LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER*8 M, N, LDA, LDB, LWORK
 REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2F( PLACE, FULL, [M], [N], A, [LDA], B, [LDB], WORK,
*       LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER :: M, N, LDA, LDB, LWORK
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE FFT2F_64( PLACE, FULL, [M], [N], A, [LDA], B, [LDB],
*       WORK, LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER(8) :: M, N, LDA, LDB, LWORK
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void rfft2f(char place, char full, int m, int n, float *a, int lda, float *b, int ldb, float *work, int lwork);

void rfft2f_64(char place, char full, long m, long n, float *a, long lda, float *b, long ldb, float *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Character. If PLACE = 'I' or 'i' (for in-place) , the input and output data are stored in array A. If PLACE = 'O' or 'o' (for out-of-place), the input data is stored in array B while the output is stored in A.
- **FULL (input)**
  Indicates whether or not to generate the full result matrix. 'F' or 'f' will cause RFFT2F to generate the full result matrix. Otherwise only a partial matrix that takes advantage of symmetry will be generated.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, a two-dimensional array `A(LDA,N)` that contains the data to be transformed. Upon exit, A is unchanged if an out-of-place transform is done. If an in-place transform with partial result is requested, `A(1:(M/2+1)*2,1:N)` will contain the transformed results. If an in-place transform with full result is requested, `A(1:2*M,1:N)` will contain complete transformed results.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA must be even if the transformed sequences are to be stored in A.

  If PLACE = ('O' or 'o') LDA > = M

  If PLACE = ('I' or 'i') LDA must be even. If

  FULL = ('F' or 'f'), LDA > = 2*M

  FULL is not ('F' or 'f'), LDA > = (M/2+1)*2
- **B (input/output)**
  Upon exit, a two-dimensional array `B(2*LDB,N)` that contains the transformed results if an out-of-place transform is done. Otherwise, B is not used.

  If an out-of-place transform is done and FULL is not 'F' or 'f', `B(1:(M/2+1)*2,1:N)` will contain the partial transformed results. If FULL = 'F' or 'f', `B(1:2*M,1:N)` will contain the complete transformed results.
- **LDB (input)**
  2*LDB is the leading dimension of the array B. If an in-place transform is desired LDB is ignored.

  If PLACE is ('O' or 'o') and

  FULL is ('F' or 'f'), LDB > = M

  FULL is not ('F' or 'f'), LDB > = M/2+1

Note that even though LDB is used in the argument list, 2*LDB is the actual leading dimension of B.

- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by RFFT2I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*N + MAX(M, 2*N) + 30)

NAME

rfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

SYNOPSIS

```
SUBROUTINE RFFT2I( M, N, WORK)
INTEGER M, N
REAL WORK(*)

SUBROUTINE RFFT2I_64( M, N, WORK)
INTEGER*8 M, N
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2I( M, N, WORK)
INTEGER :: M, N
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT2I_64( M, N, WORK)
INTEGER(8) :: M, N
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void rfft2i(int m, int n, float *work);

void rfft2i_64(long m, long n, float *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (M + 2*N + MAX(M, 2*N) + 30) or greater. RFFT2I needs to be called only once to initialize array WORK before calling RFFT2F and/or RFFT2B if M, N and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

rfft3b - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFT3F followed by a call of RFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
SUBROUTINE RFFT3B( PLACE, M, N, K, A, LDA, B, LDB, WORK, LWORK)
CHARACTER * 1 PLACE
INTEGER M, N, K, LDA, LDB, LWORK
REAL A(LDA,N,*), B(LDB,N,*), WORK(*)

SUBROUTINE RFFT3B_64( PLACE, M, N, K, A, LDA, B, LDB, WORK, LWORK)
CHARACTER * 1 PLACE
INTEGER*8 M, N, K, LDA, LDB, LWORK
REAL A(LDA,N,*), B(LDB,N,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3B( PLACE, [M], [N], [K], A, [LDA], B, [LDB], WORK,
*       LWORK)
CHARACTER(LEN=1) :: PLACE
INTEGER :: M, N, K, LDA, LDB, LWORK
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:,:) :: A, B

SUBROUTINE FFT3B_64( PLACE, [M], [N], [K], A, [LDA], B, [LDB], WORK,
*       LWORK)
CHARACTER(LEN=1) :: PLACE
INTEGER(8) :: M, N, K, LDA, LDB, LWORK
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void rfft3b(char place, int m, int n, int k, float *a, int lda, float *b, int ldb, float *work, int lwork);

void rfft3b_64(char place, long m, long n, long k, float *a, long lda, float *b, long ldb, float *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Select an in-place ('I' or 'i') or out-of-place ('O' or 'o') transform.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **K (input)**
  Integer specifying the number of planes to be transformed. It is most efficient when K is a product of small primes. K > = 0; when K = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, the three-dimensional array `A(LDA,N,K)` contains the data to be transformed if an in-place transform is requested. Otherwise, it is not referenced. Upon exit, results are stored in A(1:M,1:N,1:K).
- **LDA (input)**
  Integer specifying the leading dimension of A. If an out-of-place transform is desired LDA > = M. Else if an in-place transform is desired LDA > = 2*(M/2+1).
- **B (input/output)**
  Real array of dimension B(2*LDB,N,K). On entry, if an out-of-place transform is requested `B(1:2*(M/2+1),1:N,1:K)` contains the input data. Otherwise, B is not referenced. B is unchanged upon exit.
- **LDB (input)**
  If an out-of-place transform is desired, 2*LDB is the leading dimension of the array B which contains the data to be transformed and 2*LDB > = 2*(M/2+1). Otherwise it is not referenced.
- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. On input, WORK must have been initialized by RFFT3I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*(N + K) + 4*K + 45).

NAME

rfft3f - compute the Fourier coefficients of a real periodic sequence. The RFFT operations are unnormalized, so a call of RFFT3F followed by a call of RFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
 SUBROUTINE RFFT3F( PLACE, FULL, M, N, K, A, LDA, B, LDB, WORK,
*      LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER M, N, K, LDA, LDB, LWORK
 REAL A(LDA,N,*), B(LDB,N,*), WORK(*)

 SUBROUTINE RFFT3F_64( PLACE, FULL, M, N, K, A, LDA, B, LDB, WORK,
*      LWORK)
 CHARACTER * 1 PLACE, FULL
 INTEGER*8 M, N, K, LDA, LDB, LWORK
 REAL A(LDA,N,*), B(LDB,N,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3F( PLACE, FULL, [M], [N], [K], A, [LDA], B, [LDB],
*       WORK, LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER :: M, N, K, LDA, LDB, LWORK
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:,:) :: A, B

 SUBROUTINE FFT3F_64( PLACE, FULL, [M], [N], [K], A, [LDA], B, [LDB],
*       WORK, LWORK)
 CHARACTER(LEN=1) :: PLACE, FULL
 INTEGER(8) :: M, N, K, LDA, LDB, LWORK
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void rfft3f(char place, char full, int m, int n, int k, float *a, int lda, float *b, int ldb, float *work, int lwork);

void rfft3f_64(char place, char full, long m, long n, long k, float *a, long lda, float *b, long ldb, float *work, long lwork);

ARGUMENTS

- **PLACE (input)**
  Select an in-place ('I' or 'i') or out-of-place ('O' or 'o') transform.
- **FULL (input)**
  Select a full ('F' or 'f') or partial (' ') representation of the results. If the caller selects full representation then an MxNxK real array will transform to produce an MxNxK complex array. If the caller does not select full representation then an MxNxK real array will transform to a (M/2+1)xNxK complex array that takes advantage of the symmetry properties of a transformed real sequence.
- **M (input)**
  Integer specifying the number of rows to be transformed. It is most efficient when M is a product of small primes. M > = 0; when M = 0, the subroutine returns immediately without changing any data.
- **N (input)**
  Integer specifying the number of columns to be transformed. It is most efficient when N is a product of small primes. N > = 0; when N = 0, the subroutine returns immediately without changing any data.
- **K (input)**
  Integer specifying the number of planes to be transformed. It is most efficient when K is a product of small primes. K > = 0; when K = 0, the subroutine returns immediately without changing any data.
- **A (input/output)**
  On entry, a three-dimensional array `A(LDA,N,K)` that contains input data to be transformed. On exit, if an in-place transform is done and FULL is not 'F' or 'f', `A(1:2*(M/2+1),1:N,1:K)` will contain the partial transformed results. If FULL = 'F' or 'f', `A(1:2*M,1:N,1:K)` will contain the complete transformed results.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA must be even if the transformed sequences are to be stored in A.

  If PLACE = ('O' or 'o') LDA > = M

  If PLACE = ('I' or 'i') LDA must be even. If

  FULL = ('F' or 'f'), LDA > = 2*M

  FULL is not ('F' or 'f'), LDA > = 2*(M/2+1)
- **B (input/output)**
  Upon exit, a three-dimensional array `B(2*LDB,N,K)` that contains the transformed results if an out-of-place transform is done. Otherwise, B is not used.

  If an out-of-place transform is done and FULL is not 'F' or 'f', `B(1:2*(M/2+1),1:N,1:K)` will contain the partial transformed results. If FULL = 'F' or 'f', `B(1:2*M,1:N,1:K)` will contain the complete transformed results.
- **LDB (input)**
  2*LDB is the leading dimension of the array B. If an in-place transform is desired LDB is ignored.

  If PLACE is ('O' or 'o') and

FULL is ('F' or 'f'), then LDB > = M

FULL is not ('F' or 'f'), then LDB > = M/2 + 1

Note that even though LDB is used in the argument list, 2*LDB is the actual leading dimension of B.

- **WORK (input/output)**
  One-dimensional real array of length at least LWORK. WORK must have been initialized by RFFT3I.
- **LWORK (input)**
  Integer. LWORK > = (M + 2*(N + K) + 4*K + 45).

NAME

rfft3i - initialize the array WSAVE, which is used in both RFFT3F and RFFT3B.

SYNOPSIS

```
SUBROUTINE RFFT3I( M, N, K, WORK)
INTEGER M, N, K
REAL WORK(*)

SUBROUTINE RFFT3I_64( M, N, K, WORK)
INTEGER*8 M, N, K
REAL WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3I( M, N, K, WORK)
INTEGER :: M, N, K
REAL, DIMENSION(:) :: WORK

SUBROUTINE FFT3I_64( M, N, K, WORK)
INTEGER(8) :: M, N, K
REAL, DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void rfft3i(int m, int n, int k, float *work);

void rfft3i_64(long m, long n, long k, float *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **K (input)**
  Number of planes to be transformed. K > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (M + 2*(N + K) + 4*K + 45) or greater. RFFT3I needs to be called only once to initialize array WORK before calling RFFT3F and/or RFFT3B if M, N, K and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

rfftb - compute a periodic sequence from its Fourier coefficients. The RFFT operations are unnormalized, so a call of RFFTF followed by a call of RFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE RFFTB( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE RFFTB_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [N], X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE FFTB_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void rfftb(int n, float *x, float *wsave);

void rfftb_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (2 * N + 15) or greater and must have been initialized by RFFTI.

NAME

rfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of RFFTF followed by a call of RFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE RFFTF( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE RFFTF_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [N], X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE FFTF_64( [N], X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void rfftf(int n, float *x, float *wsave);

void rfftf_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (2 * N + 15) or greater and must have been initialized by RFFTI.

NAME

rffti - initialize the array WSAVE, which is used in both RFFTF and RFFTB.

SYNOPSIS

```
SUBROUTINE RFFTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE RFFTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE FFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void rffti(int n, float *wsave);

void rffti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. RFFTI needs to be called only once to initialize array WORK before calling RFFTF and/or RFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

rfftopt - compute the length of the closest fast FFT

SYNOPSIS

```
INTEGER FUNCTION RFFTOPT( LEN)
INTEGER LEN

INTEGER*8 FUNCTION RFFTOPT_64( LEN)
INTEGER*8 LEN
```

F95 INTERFACE

```
INTEGER FUNCTION RFFTOPT( LEN)
INTEGER :: LEN

INTEGER(8) FUNCTION RFFTOPT_64( LEN)
INTEGER(8) :: LEN
```

C INTERFACE

#include <sunperf.h>

int rfftopt(int len);

long rfftopt_64(long len);

PURPOSE

rfftopt computes the length of the closest fast FFT. Fast Fourier transform algorithms, including those used in Performance Library, work best with vector lengths that are products of small primes. For example, an FFT of length 32=2**5 will run faster than an FFT of prime length 31 because 32 is a product of small primes and 31 is not. If your application is such that you can taper or zero pad your vector to a larger length then this function may help you select a better length and run your FFT faster.

RFFTOPT will return an integer no smaller than the input argument N that is the closest number that is the product of small primes. RFFTOPT will return 16 for an input of N=16 and return 18=2*3*3 for an input of N=17.

Note that the length computed here is not guaranteed to be optimal, only to be a product of small primes. Also, the value returned may change as the underlying FFTs become capable of handling larger primes. For example, passing in N=51 to day will return 52=2*2*13 rather than 51=3*17 because the FFTs in Performance Li brary do not have fast radix 17 code. In the future, radix 17 code may be added and then N=51 will return 51.

ARGUMENTS

NAME

sasum - Return the sum of the absolute values of a vector x.

SYNOPSIS

```
REAL FUNCTION SASUM( N, X, INCX)
INTEGER N, INCX
REAL X(*)

REAL FUNCTION SASUM_64( N, X, INCX)
INTEGER*8 N, INCX
REAL X(*)
```

F95 INTERFACE

```
REAL FUNCTION ASUM( [N], X, [INCX])
INTEGER :: N, INCX
REAL, DIMENSION(:) :: X

REAL FUNCTION ASUM_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

float sasum(int n, float *x, int incx);

float sasum_64(long n, float *x, long incx);

PURPOSE

sasum Return the sum of the absolute values of x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.

NAME

saxpy - compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE SAXPY( N, ALPHA, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
REAL ALPHA
REAL X(*), Y(*)

SUBROUTINE SAXPY_64( N, ALPHA, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
REAL ALPHA
REAL X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE AXPY( [N], ALPHA, X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y

SUBROUTINE AXPY_64( [N], ALPHA, X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void saxpy(int n, float alpha, float *x, int incx, float *y, int incy);

void saxpy_64(long n, float alpha, float *x, long incx, float *y, long incy);

PURPOSE

saxpy compute y := alpha * x + y where alpha is a scalar and x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

saxpyi - Compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE SAXPYI(NZ, A, X, INDX, Y)

REAL A
REAL X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE SAXPYI_64(NZ, A, X, INDX, Y)

REAL A
REAL X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE AXPYI([NZ], [A], X, INDX, Y)

```
REAL :: A
REAL, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE AXPYI_64([NZ], [A], X, INDX, Y)

REAL :: A
REAL, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

SAXPYI Compute y := alpha * x + y where alpha is a scalar, x is a sparse vector, and y is a vector in full storage form

```
do i = 1, n
  y(indx(i)) = alpha * x(i) + y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

A (input)

On entry, ALPHA specifies the scaling value. Unchanged on exit.

X (input)

Vector containing the values of the compressed form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

NAME

sbdsdc - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B

SYNOPSIS

```
 SUBROUTINE SBDSDC( UPLO, COMPQ, N, D, E, U, LDU, VT, LDVT, Q, IQ,
*      WORK, IWORK, INFO)
 CHARACTER * 1 UPLO, COMPQ
 INTEGER N, LDU, LDVT, INFO
 INTEGER IQ(*), IWORK(*)
 REAL D(*), E(*), U(LDU,*), VT(LDVT,*), Q(*), WORK(*)

 SUBROUTINE SBDSDC_64( UPLO, COMPQ, N, D, E, U, LDU, VT, LDVT, Q, IQ,
*      WORK, IWORK, INFO)
 CHARACTER * 1 UPLO, COMPQ
 INTEGER*8 N, LDU, LDVT, INFO
 INTEGER*8 IQ(*), IWORK(*)
 REAL D(*), E(*), U(LDU,*), VT(LDVT,*), Q(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSDC( UPLO, COMPQ, [N], D, E, U, [LDU], VT, [LDVT], Q,
*       IQ, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO, COMPQ
 INTEGER :: N, LDU, LDVT, INFO
 INTEGER, DIMENSION(:) :: IQ, IWORK
 REAL, DIMENSION(:) :: D, E, Q, WORK
 REAL, DIMENSION(:,:) :: U, VT

 SUBROUTINE BDSDC_64( UPLO, COMPQ, [N], D, E, U, [LDU], VT, [LDVT],
*       Q, IQ, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO, COMPQ
 INTEGER(8) :: N, LDU, LDVT, INFO
 INTEGER(8), DIMENSION(:) :: IQ, IWORK
 REAL, DIMENSION(:) :: D, E, Q, WORK
 REAL, DIMENSION(:,:) :: U, VT
```

C INTERFACE

#include <sunperf.h>

void sbdsdc(char uplo, char compq, int n, float *d, float *e, float *u, int ldu, float *vt, int ldvt, float *q, int *iq, int *info);

void sbdsdc_64(char uplo, char compq, long n, float *d, float *e, float *u, long ldu, float *vt, long ldvt, float *q, long *iq, long *info);

PURPOSE

sbdsdc computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = U * S * VT, using a divide and conquer method, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and VT are orthogonal matrices of left and right singular vectors, respectively. SBDSDC can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLASD3 for details.

The code currently call SLASDQ if singular values only are desired. However, it can be slightly modified to compute singular values using the divide and conquer method.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  B is upper bidiagonal.

= 'L':  B is lower bidiagonal.
```

- **COMPQ (input)**
  Specifies whether singular vectors are to be computed as follows:

```
= 'N':  Compute singular values only;

= 'P':  Compute singular values and compute singular
vectors in compact form;
= 'I':  Compute singular values and singular vectors.
```

- **N (input)**
  The order of the matrix B. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of the bidiagonal matrix whose SVD is desired. On exit, E has been destroyed.
- **U (output)**

If COMPQ = 'I', then: On exit, if INFO = 0, U contains the left singular vectors of the bidiagonal matrix. For other values of COMPQ, U is not referenced.

- **LDU (input)**
  The leading dimension of the array U. LDU > = 1. If singular vectors are desired, then LDU > = max(1, N).
- **VT (output)**
  If COMPQ = 'I', then: On exit, if INFO = 0, VT' contains the right singular vectors of the bidiagonal matrix. For other values of COMPQ, VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT > = 1. If singular vectors are desired, then LDVT > = max(1, N).
- **Q (output)**
  If COMPQ = 'P', then: On exit, if INFO = 0, Q and IQ contain the left and right singular vectors in a compact form, requiring O(N log N) space instead of 2*N**2. In particular, Q contains all the REAL data in LDQ > = N*(11 + 2*SMLSIZ + 8*INT(LOG_2(N/(SMLSIZ+1)))) words of memory, where SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25). For other values of COMPQ, Q is not referenced.
- **IQ (output)**
  If COMPQ = 'P', then: On exit, if INFO = 0, Q and IQ contain the left and right singular vectors in a compact form, requiring O(N log N) space instead of 2*N**2. In particular, IQ contains all INTEGER data in LDIQ > = N*(3 + 3*INT(LOG_2(N/(SMLSIZ+1)))) words of memory, where SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25). For other values of COMPQ, IQ is not referenced.
- **WORK (workspace)**
  If COMPQ = 'N' then LWORK > = (2 * N). If COMPQ = 'P' then LWORK > = (6 * N). If COMPQ = 'I' then LWORK > = (3 * N**2 + 4 * N).
- **IWORK (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an singular value.
The update process of divide and conquer failed.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Huan Ren, Computer Science Division, University of
California at Berkeley, USA
```

NAME

sbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

SYNOPSIS

```
 SUBROUTINE SBDSQR( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U, LDU,
*      C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL D(*), E(*), VT(LDVT,*), U(LDU,*), C(LDC,*), WORK(*)

 SUBROUTINE SBDSQR_64( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U,
*      LDU, C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL D(*), E(*), VT(LDVT,*), U(LDU,*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSQR( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT, [LDVT],
*       U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: VT, U, C

 SUBROUTINE BDSQR_64( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT,
*       [LDVT], U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: VT, U, C
```

C INTERFACE

#include <sunperf.h>

void sbdsqr(char uplo, int n, int ncvt, int nru, int ncc, float *d, float *e, float *vt, int ldvt, float *u, int ldu, float *c, int ldc, int *info);

void sbdsqr_64(char uplo, long n, long ncvt, long nru, long ncc, float *d, float *e, float *vt, long ldvt, float *u, long ldu, float *c, long ldc, long *info);

PURPOSE

sbdsqr computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = Q * S * P' (P' denotes the transpose of P), where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and Q and P are orthogonal matrices.

The routine computes S, and optionally computes U * Q, P' * VT, or Q' * C, for given real input matrices U, VT, and C.

See ``Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy," by J. Demmel and W. Kahan, LAPACK Working Note #3 (or SIAM J. Sci. Statist. Comput. vol. 11, no. 5, pp. 873-912, Sept 1990) and

``Accurate singular values and differential qd algorithms," by B. Parlett and V. Fernando, Technical Report CPAM-554, Mathematics Department, University of California at Berkeley, July 1992 for a detailed description of the algorithm.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  B is upper bidiagonal;

  = 'L':  B is lower bidiagonal.
  ```

- **N (input)**
  The order of the matrix B. N > = 0.
- **NCVT (input)**
  The number of columns of the matrix VT. NCVT > = 0.
- **NRU (input)**
  The number of rows of the matrix U. NRU > = 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B in decreasing order.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of the bidiagonal matrix whose SVD is desired. On normal exit (INFO = 0), E is destroyed. If the algorithm does not converge (INFO > 0), D and E will contain the diagonal and superdiagonal elements of a bidiagonal matrix orthogonally equivalent to the one given as input. E(N) is used for workspace.
- **VT (input/output)**

On entry, an N-by-NCVT matrix VT. On exit, VT is overwritten by P' * VT. VT is not referenced if NCVT = 0.

- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= `max(1,N)` if NCVT > 0; LDVT >= 1 if NCVT = 0.
- **U (input/output)**
  On entry, an NRU-by-N matrix U. On exit, U is overwritten by U * Q. U is not referenced if NRU = 0.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,NRU).
- **C (input/output)**
  On entry, an N-by-NCC matrix C. On exit, C is overwritten by Q' * C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,N)` if NCC > 0; LDC >=1 if NCC = 0.
- **WORK (workspace)**
  `dimension(4*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  If INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm did not converge; D and E contain the
elements of a bidiagonal matrix which is orthogonally
similar to the input matrix B;  if INFO  = i, i
elements of E have not converged to zero.
```

NAME

scasum - Return the sum of the absolute values of a vector x.

SYNOPSIS

```
REAL FUNCTION SCASUM( N, X, INCX)
COMPLEX X(*)
INTEGER N, INCX

REAL FUNCTION SCASUM_64( N, X, INCX)
COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
REAL FUNCTION ASUM( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER :: N, INCX

REAL FUNCTION ASUM_64( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

float scasum(int n, complex *x, int incx);

float scasum_64(long n, complex *x, long incx);

PURPOSE

scasum Return the sum of the absolute values of the elements of x where x is an n-vector. This is the sum of the absolute values of the real and complex elements and not the sum of the squares of the real and complex elements.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.

NAME

scnrm2 - Return the Euclidian norm of a vector.

SYNOPSIS

```
REAL FUNCTION SCNRM2( N, X, INCX)
COMPLEX X(*)
INTEGER N, INCX

REAL FUNCTION SCNRM2_64( N, X, INCX)
COMPLEX X(*)
INTEGER*8 N, INCX
```

F95 INTERFACE

```
REAL FUNCTION NRM2( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER :: N, INCX

REAL FUNCTION NRM2_64( [N], X, [INCX])
COMPLEX, DIMENSION(:) :: X
INTEGER(8) :: N, INCX
```

C INTERFACE

#include <sunperf.h>

float scnrm2(int n, complex *x, int incx);

float scnrm2_64(long n, complex *x, long incx);

PURPOSE

scnrm2 Return the Euclidian norm of a vector x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NAME

scnvcor - compute the convolution or correlation of real vectors

---

SYNOPSIS

```
 SUBROUTINE SCNVCOR( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M, Y,
*       IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 INTEGER NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL X(*), Y(*), Z(*), WORK(*)

 SUBROUTINE SCNVCOR_64( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M,
*       Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 INTEGER*8 NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL X(*), Y(*), Z(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE, M,
*        Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 INTEGER :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL, DIMENSION(:) :: X, Y, Z, WORK

 SUBROUTINE CNVCOR_64( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE,
*        M, Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK,
*        [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 INTEGER(8) :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
 REAL, DIMENSION(:) :: X, Y, Z, WORK
```

C INTERFACE

#include <sunperf.h>

void scnvcor(char cnvcor, char four, int nx, float *x, int ifx, int incx, int ny, int npre, int m, float *y, int ify, int inc1y, int inc2y, int nz, int k, float *z, int ifz, int inc1z, int inc2z, float *work, int lwork);

void scnvcor_64(char cnvcor, char four, long nx, float *x, long ifx, long incx, long ny, long npre, long m, float *y, long ify, long inc1y, long inc2y, long nz, long k, float *z, long ifz, long inc1z, long inc2z, float *work, long lwork);

PURPOSE

scnvcor computes the convolution or correlation of real vectors.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' if convolution is desired, 'R' or 'r' if correlation is desired.
- **FOUR (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' if the computation should be done directly from the definition. The Fourier transform method is generally faster, but it may introduce noticeable errors into certain results, notably when both the filter and data vectors consist entirely of integers or vectors where elements of either the filter vector or a given data vector differ significantly in magnitude from the 1-norm of the vector.
- **NX (input)**
  Length of the filter vector. NX > = 0. SCNVCOR will return immediately if NX = 0.
- **X (input)**
  Filter vector.
- **IFX (input)**
  Index of the first element of X. NX > = IFX > = 1.
- **INCX (input)**
  Stride between elements of the filter vector in X. INCX > 0.
- **NY (input)**
  Length of the input vectors. NY > = 0. SCNVCOR will return immediately if NY = 0.
- **NPRE (input)**
  The number of implicit zeros prepended to the Y vectors. NPRE > = 0.
- **M (input)**
  Number of input vectors. M > = 0. SCNVCOR will return immediately if M = 0.
- **Y (input)**
  Input vectors.
- **IFY (input)**
  Index of the first element of Y. NY > = IFY > = 1.
- **INC1Y (input)**
  Stride between elements of the input vectors in Y. INC1Y > 0.
- **INC2Y (input)**
  Stride between the input vectors in Y. INC2Y > 0.
- **NZ (input)**
  Length of the output vectors. NZ > = 0. SCNVCOR will return immediately if NZ = 0. See the Notes section below for information about how this argument interacts with NX and NY to control circular versus end-off shifting.
- **K (input)**
  Number of Z vectors. K > = 0. If K = 0 then SCNVCOR will return immediately. If K < M then only the first K input vectors will be processed. If K > M then M input vectors will be processed.
- **Z (output)**
  Result vectors.
- **IFZ (input)**
  Index of the first element of Z. NZ > = IFZ > = 1.
- **INC1Z (input)**
  Stride between elements of the output vectors in Z. INC1Z > 0.
- **INC2Z (input)**
  Stride between the output vectors in Z. INC2Z > 0.
- **WORK (input/output)**
  Scratch space. Before the first call to SCNVCOR with particular values of the integer arguments the first element of WORK must be set to zero. If WORK is written between calls to SCNVCOR or if SCNVCOR is called with different values of the integer arguments then the first element of WORK must again be set to zero before each call. If WORK has not been written and the same

values of the integer arguments are used then the first element of WORK to zero. This can avoid certain initializations that store their results into WORK, and avoiding the initialization can make SCNVCOR run faster.

- **LWORK (input)**
  Length of WORK. LWORK > = 4*MAX(NX,NY,NZ)+15. =head1 NOTES If any vector overlaps a writable vector, either because of argument aliasing or ill-chosen values of the various INC arguments, the results are undefined and may vary from one run to the next.

  The most common form of the computation, and the case that executes fastest, is applying a filter vector X to a series of vectors stored in the columns of Y with the result placed into the columns of Z. In that case, INCX = 1, INC1Y = 1, INC2Y > = NY, INC1Z = 1, INC2Z > = NZ. Another common form is applying a filter vector X to a series of vectors stored in the rows of Y and store the result in the row of Z, in which case INCX = 1, INC1Y > = NY, INC2Y = 1, INC1Z > = NZ, and INC2Z = 1.

  A common use of convolution is to compute the products of polynomials. The following code uses SCNVCOR to compute the product of 1 + 2x + 3x**2 and 4 + 5x + 6x**2:

NAME

scnvcor2 - compute the convolution or correlation of real matrices

---

SYNOPSIS

```
 SUBROUTINE SCNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*      SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*      LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX WORKIN(*)
 INTEGER MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL X(LDX,*), Y(LDY,*), Z(LDZ,*)

 SUBROUTINE SCNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*      SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*      LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX WORKIN(*)
 INTEGER*8 MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL X(LDX,*), Y(LDY,*), Z(LDZ,*)
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX, DIMENSION(:) :: WORKIN
 INTEGER :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL, DIMENSION(:,:) :: X, Y, Z

 SUBROUTINE CNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX, DIMENSION(:) :: WORKIN
 INTEGER(8) :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
 REAL, DIMENSION(:,:) :: X, Y, Z
```

C INTERFACE

#include <sunperf.h>

void scnvcor2(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, int mx, int nx, float *x, int ldx, int my, int ny, int mpre, int npre, float *y, int ldy, int mz, int nz, float *z, int ldz, complex *workin, int lwork);

void scnvcor2_64(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, long mx, long nx, float *x, long ldx, long my, long ny, long mpre, long npre, float *y, long ldy, long mz, long nz, float *z, long ldz, complex *workin, long lwork);

PURPOSE

scnvcor2 computes the convolution or correlation of real matrices.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' to compute convolution, 'R' or 'r' to compute correlation.
- **METHOD (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' to compute directly from the definition.
- **TRANSX (input)**
  \'N' or 'n' if X is the filter matrix, 'T' or 't' if `transpose(X)` is the filter matrix.
- **SCRATCHX (input)**
  \'N' or 'n' if X must be preserved, 'S' or 's' if X can be used as scratch space. The contents of X are undefined after returning from a call in which X is allowed to be used for scratch.
- **TRANSY (input)**
  \'N' or 'n' if Y is the input matrix, 'T' or 't' if `transpose(Y)` is the input matrix.
- **SCRATCHY (input)**
  \'N' or 'n' if Y must be preserved, 'S' or 's' if Y can be used as scratch space. The contents of Y are undefined after returning from a call in which Y is allowed to be used for scratch.
- **MX (input)**
  Number of rows in the filter matrix. MX > = 0.
- **NX (input)**
  Number of columns in the filter matrix. NX > = 0.
- **X (input)**

  ```
  dimension(LDX,NX)
  ```

  On entry, the filter matrix. Unchanged on exit if SCRATCHX is 'N' or 'n', undefined on exit if SCRATCHX is 'S' or 's'.
- **LDX (input)**
  Leading dimension of the array that contains the filter matrix.
- **MY (input)**
  Number of rows in the input matrix. MY > = 0.

- **NY (input)**
  Number of columns in the input matrix. NY > = 0.
- **MPRE (input)**
  Number of implicit zeros to prepend to each row of the input matrix. MPRE > = 0.
- **NPRE (input)**
  Number of implicit zeros to prepend to each column of the input matrix. NPRE > = 0.
- **Y (input)**

  ```
  dimension(LDY,*)
  ```

  Input matrix. Unchanged on exit if SCRATCHY is 'N' or 'n', undefined on exit if SCRATCHY is 'S' or 's'.
- **LDY (input)**
  Leading dimension of the array that contains the input matrix.
- **MZ (input)**
  Number of rows in the output matrix. MZ > = 0. SCNVCOR2 will return immediately if MZ = 0.
- **NZ (input)**
  Number of columns in the output matrix. NZ > = 0. SCNVCOR2 will return immediately if NZ = 0.
- **Z (output)**

  ```
  dimension(LDZ,*)
  ```

  Result matrix.
- **LDZ (input)**
  Leading dimension of the array that contains the result matrix. LDZ > = MAX(1,MZ).
- **WORKIN (input/output)**
  (input/scratch) `dimension(LWORK)`

  On entry for the first call to SCNVCOR2, `WORKIN(1)` must contain 0.0. After the first call, `WORKIN(1)` must be set to 0.0 iff WORKIN has been altered since the last call to this subroutine or if the sizes of the arrays have changed.
- **LWORK (input)**
  Length of the work vector. If the FFT is to be used then for best performance LWORK should be at least 30 words longer than the amount of memory needed to hold the trig tables. If the FFT is not used, the value of LWORK is unimportant.

NAME

scopy - Copy x to y

SYNOPSIS

```
SUBROUTINE SCOPY( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
REAL X(*), Y(*)

SUBROUTINE SCOPY_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
REAL X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE COPY( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y

SUBROUTINE COPY_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void scopy(int n, float *x, int incx, float *y, int incy);

void scopy_64(long n, float *x, long incx, float *y, long incy);

PURPOSE

scopy Copy x to y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

sdisna - compute the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix

---

SYNOPSIS

```
SUBROUTINE SDISNA( JOB, M, N, D, SEP, INFO)
CHARACTER * 1 JOB
INTEGER M, N, INFO
REAL D(*), SEP(*)

SUBROUTINE SDISNA_64( JOB, M, N, D, SEP, INFO)
CHARACTER * 1 JOB
INTEGER*8 M, N, INFO
REAL D(*), SEP(*)
```

F95 INTERFACE

```
SUBROUTINE DISNA( JOB, [M], N, D, SEP, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER :: M, N, INFO
REAL, DIMENSION(:) :: D, SEP

SUBROUTINE DISNA_64( JOB, [M], N, D, SEP, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER(8) :: M, N, INFO
REAL, DIMENSION(:) :: D, SEP
```

C INTERFACE

#include <sunperf.h>

void sdisna(char job, int m, int n, float *d, float *sep, int *info);

void sdisna_64(char job, long m, long n, float *d, float *sep, long *info);

PURPOSE

sdisna computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m-by-n matrix. The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the I-th computed vector is given by

```
SLAMCH( 'E' ) * ( ANORM / SEP( I ) )
```

where ANORM = 2-norm(A) = max(abs(D(j))). SEP(I) is not allowed to be smaller than SLAMCH('E')*ANORM in order to limit the size of the error bound.

SDISNA may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

ARGUMENTS

- **JOB (input)**
  Specifies for which problem the reciprocal condition numbers should be computed:

  ```
  = 'E':  the eigenvectors of a symmetric/Hermitian matrix;

  = 'L':  the left singular vectors of a general matrix;

  = 'R':  the right singular vectors of a general matrix.
  ```
- **M (input)**
  The number of rows of the matrix. M >= 0.
- **N (input)**
  If JOB = 'L' or 'R', the number of columns of the matrix, in which case N >= 0. Ignored if JOB = 'E'.
- **D (input)**
  dimension (min(M,N)) if JOB = 'L' or 'R' The eigenvalues (if JOB = 'E') or singular values (if JOB = 'L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.
- **SEP (output)**
  dimension (min(M,N)) if JOB = 'L' or 'R' The reciprocal condition numbers of the vectors.
- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

sdot - compute the dot product of two vectors x and y.

SYNOPSIS

```
REAL FUNCTION SDOT( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
REAL X(*), Y(*)

REAL FUNCTION SDOT_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
REAL X(*), Y(*)
```

F95 INTERFACE

```
REAL FUNCTION DOT( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y

REAL FUNCTION DOT_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

float sdot(int n, float *x, int incx, float *y, int incy);

float sdot_64(long n, float *x, long incx, float *y, long incy);

PURPOSE

sdot compute the dot product of x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

sdoti - Compute the indexed dot product.

SYNOPSIS

```
REAL FUNCTION SDOTI(NZ, X, INDX, Y)

REAL X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

REAL FUNCTION SDOTI_64(NZ, X, INDX, Y)

REAL X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE REAL FUNCTION DOTI([NZ], X, INDX, Y)

```
REAL, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

REAL FUNCTION DOTI_64([NZ], X, INDX, Y)

REAL, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

SDOTI Compute the indexed dot product of a real sparse vector x stored in compressed form with a real vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + x(i) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

sdsdot - compute a constant plus the double precision dot product of two single precision vectors x and y

---

SYNOPSIS

```
REAL FUNCTION SDSDOT( N, SB, SX, INCX, SY, INCY)
INTEGER N, INCX, INCY
REAL SB
REAL SX(*), SY(*)

REAL FUNCTION SDSDOT_64( N, SB, SX, INCX, SY, INCY)
INTEGER*8 N, INCX, INCY
REAL SB
REAL SX(*), SY(*)
```

F95 INTERFACE

```
REAL FUNCTION SDSDOT( N, SB, SX, INCX, SY, INCY)
INTEGER :: N, INCX, INCY
REAL :: SB
REAL, DIMENSION(:) :: SX, SY

REAL FUNCTION SDSDOT_64( N, SB, SX, INCX, SY, INCY)
INTEGER(8) :: N, INCX, INCY
REAL :: SB
REAL, DIMENSION(:) :: SX, SY
```

C INTERFACE

#include <sunperf.h>

float sdsdot(int n, float sb, float *sx, int incx, float *sy, int incy);

float sdsdot_64(long n, float sb, float *sx, long incx, float *sy, long incy);

PURPOSE

sdsdot Computes a constant plus the double precision dot product of x and y where x and y are single precision n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **SB (input)**
  On entry, the constant that is added to the dot product before the result is returned. Unchanged on exit.
- **SX (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array SX must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of SX. INCX must not be zero. Unchanged on exit.
- **SY (input)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array SY must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of SY. INCY must not be zero. Unchanged on exit.

NAME

second - return the user time for a process in seconds =head1 SYNOPSIS

```
REAL FUNCTION SECOND( )

REAL FUNCTION SECOND_64( )
```

F95 INTERFACE

```
REAL FUNCTION SECOND( )

REAL FUNCTION SECOND_64( )
```

C INTERFACE

#include <sunperf.h>

float second();

float second_64();

PURPOSE

second returns the user time for a process in seconds. This version gets the time from the system function ETIME.

ARGUMENTS

NAME

sfftc - initialize the trigonometric weight and factor tables or compute the forward Fast Fourier Transform of a real sequence.
=head1 SYNOPSIS

```
 SUBROUTINE SFFTC( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX Y(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL X(*), TRIGS(*), WORK(*)

 SUBROUTINE SFFTC_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 COMPLEX Y(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL X(*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX, DIMENSION(:) :: Y
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: X, TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK,
*       [LWORK], IERR)
 COMPLEX, DIMENSION(:) :: Y
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: X, TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void sfftc(int iopt, int n, float scale, float *x, complex *y, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void sfftc_64(long iopt, long n, float scale, float *x, complex *y, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

sfftc initializes the trigonometric weight and factor tables or computes the forward Fast Fourier Transform of a real sequence as follows: .Ve

```
                N-1
```

Y(k) = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

`k ranges from 0 to N-1`

i = `sqrt(-1)`

isign = -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

In real-to-complex transform of length N, the (N/2+1) complex output data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**

On entry, X is a real array whose first N elements contain the sequence to be transformed.

- **Y (output)**
  Complex array whose first (N/2+1) elements contain the transform results. X and Y may be the same array starting at the same memory location, in which case the dimension of X must be at least 2*(N/2+1). Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **TRIGS (input/output)**
  Real array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

sfftc2 - initialize the trigonometric weight and factor tables or compute the two-dimensional forward Fast Fourier Transform of a two-dimensional real array. =head1 SYNOPSIS

```
 SUBROUTINE SFFTC2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL X(LDX,*), TRIGS(*), WORK(*)

 SUBROUTINE SFFTC2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL X(LDX,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: X

 SUBROUTINE FFT2_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void sfftc2(int iopt, int n1, int n2, float scale, float *x, int ldx, complex *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void sfftc2_64(long iopt, long n1, long n2, float scale, float *x, long ldx, complex *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

sfftc2 initializes the trigonometric weight and factor tables or computes the two-dimensional forward Fast Fourier Transform of a two-dimensional real array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the columns of the input array. One-dimensional FFTs are then computed along the rows of the intermediate results. .Ve

```
                    N2-1  N1-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

In real-to-complex transform of length N1, the (N1/2+1) complex output data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.

- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes N2 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a complex array of dimensions (LDX, N2) that contains input data to be transformed. X and Y can be the same array.
- **LDX (input)**
  Leading dimension of X. LDX > = N1 if X is not the same array as Y. Else, LDX = 2*LDY. Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. LDY > = N1/2+1 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least MAX(N1, 2*N2). The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1) or (LDX not equal 2*LDY when X and Y are same array)

  -5 = (LDY < N1/2+1)

  -6 = (LWORK not equal 0) and (LWORK < `MAX(N1,2*N2))`

  -7 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output array Y(1:LDY, 1:N2) is overwritten.

NAME

sfftc3 - initialize the trigonometric weight and factor tables or compute the three-dimensional forward Fast Fourier Transform of a three-dimensional complex array. =head1 SYNOPSIS

```
 SUBROUTINE SFFTC3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX Y(LDY1,LDY2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL X(LDX1,LDX2,*), TRIGS(*), WORK(*)

 SUBROUTINE SFFTC3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 COMPLEX Y(LDY1,LDY2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL X(LDX1,LDX2,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: Y
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:,:) :: X

 SUBROUTINE FFT3_64( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1],
*       LDX2, Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:,:) :: Y
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void sfftc3(int iopt, int n1, int n2, int n3, float scale, float *x, int ldx1, int ldx2, complex *y, int ldy1, int ldy2, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void sfftc3_64(long iopt, long n1, long n2, long n3, float scale, float *x, long ldx1, long ldx2, complex *y, long ldy1, long ldy2, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

sfftc3 initializes the trigonometric weight and factor tables or computes the three-dimensional forward Fast Fourier Transform of a three-dimensional complex array. .Ve

```
                        N3-1  N2-1  N1-1
```

`Y(k1,k2,k3)` = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                        j3=0  j2=0  j1=0
```

`.Ve`

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Real scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a real array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed. X can be same array as Y.
- **LDX1 (input)**
  first dimension of X. If X is not same array as Y, LDX1 > = N1 Else, LDX1 = 2*LDY1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. LDY1 > = N1/2+1 Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least (MAX(N,2*N2,2*N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1) or (LDX not equal 2*LDY when X and Y are same array)

  -6 = (LDX2 < N2)

  -7 = (LDY1 < N1/2+1)

  -8 = (LDY2 < N2) or (LDY2 not equal LDX2 when X and Y are same array)

  -9 = (LWORK not equal 0) and (LWORK < (MAX(N,2*N2,2*N3) + 16*N3)*NCPUS)

  -10 = memory allocation failed

SEE ALSO

fft

CAUTIONS

This routine uses `Y((N1/2+1)+1:LDY1,:,:)` as scratch space. Therefore, the original contents of this subarray will be lost upon returning from routine while subarray `Y(1:N1/2+1,1:N2,1:N3)` contains the transform results.

NAME

sfftcm - initialize the trigonometric weight and factor tables or compute the one-dimensional forward Fast Fourier Transform of a set of real data sequences stored in a two-dimensional array. =head1 SYNOPSIS

```
 SUBROUTINE SFFTCM( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 COMPLEX Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 REAL SCALE
 REAL X(LDX,*), TRIGS(*), WORK(*)

 SUBROUTINE SFFTCM_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 COMPLEX Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 REAL SCALE
 REAL X(LDX,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: X

 SUBROUTINE FFTM_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX, DIMENSION(:,:) :: Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL :: SCALE
 REAL, DIMENSION(:) :: TRIGS, WORK
 REAL, DIMENSION(:,:) :: X
```

C INTERFACE

#include <sunperf.h>

void sfftcm(int iopt, int n1, int n2, float scale, float *x, int ldx, complex *y, int ldy, float *trigs, int *ifac, float *work, int lwork, int *ierr);

void sfftcm_64(long iopt, long n1, long n2, float scale, float *x, long ldx, complex *y, long ldy, float *trigs, long *ifac, float *work, long lwork, long *ierr);

PURPOSE

sfftcm initializes the trigonometric weight and factor tables or computes the one-dimensional forward Fast Fourier Transform of a set of real data sequences stored in a two-dimensional array: .Ve

```
                    N1-1
```

Y(k,l) = scale * SUM W*X(j,l)

```
                    j=0
```

.Ve

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N1)`

In real-to-complex transform of length N1, the (N1/2+1) complex output data points stored are the positive-frequency half of the spectrum of the discrete Fourier transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT
- **N1 (input)**
  Integer specifying length of the input sequences. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying number of input sequences. N2 > = 0. Unchanged on exit.
- **SCALE (input)**

Real scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  X is a real array of dimensions (LDX, N2) that contains the sequences to be transformed stored in its columns.
- **LDX (input)**
  Leading dimension of X. If X and Y are the same array, LDX = 2*LDY Else LDX > = N1 Unchanged on exit.
- **Y (output)**
  Y is a complex array of dimensions (LDY, N2) that contains the transform results of the input sequences. X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. LDY > = N1/2 + 1 Unchanged on exit.
- **TRIGS (input/output)**
  Real array of length 2*N1 that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N1. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Real array of dimension at least N1. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1) or (LDX not equal 2*LDY when X and Y are same array)

  -4 = (LDY < N1/2 + 1)

  -6 = (LWORK not equal 0) and (LWORK < N1)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

sgbbrd - reduce a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation

---

SYNOPSIS

```
 SUBROUTINE SGBBRD( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q, LDQ,
*      PT, LDPT, C, LDC, WORK, INFO)
 CHARACTER * 1 VECT
 INTEGER M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL AB(LDAB,*), D(*), E(*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)

 SUBROUTINE SGBBRD_64( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q,
*      LDQ, PT, LDPT, C, LDC, WORK, INFO)
 CHARACTER * 1 VECT
 INTEGER*8 M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL AB(LDAB,*), D(*), E(*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBBRD( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D, E,
*       Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 INTEGER :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: AB, Q, PT, C

 SUBROUTINE GBBRD_64( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D,
*       E, Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 INTEGER(8) :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: AB, Q, PT, C
```

C INTERFACE

#include <sunperf.h>

void sgbbrd(char vect, int m, int n, int ncc, int kl, int ku, float *ab, int ldab, float *d, float *e, float *q, int ldq, float *pt, int ldpt, float *c, int ldc, int *info);

void sgbbrd_64(char vect, long m, long n, long ncc, long kl, long ku, float *ab, long ldab, float *d, float *e, float *q, long ldq, float *pt, long ldpt, float *c, long ldc, long *info);

PURPOSE

sgbbrd reduces a real general m-by-n band matrix A to upper bidiagonal form B by an orthogonal transformation: Q' * A * P = B.

The routine computes B, and optionally forms Q or P', or computes Q'*C for a given matrix C.

ARGUMENTS

- **VECT (input)**
  Specifies whether or not the matrices Q and P' are to be formed. = 'N': do not form Q or P';

  ```
  = 'Q': form Q only;

  = 'P': form P' only;

  = 'B': form both.
  ```
- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC >= 0.
- **KL (input)**
  The number of subdiagonals of the matrix A. KL >= 0.
- **KU (input)**
  The number of superdiagonals of the matrix A. KU >= 0.
- **AB (input/output)**
  On entry, the m-by-n band matrix A, stored in rows 1 to KL+KU+1. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl). On exit, A is overwritten by values generated during the reduction.
- **LDAB (input)**
  The leading dimension of the array A. LDAB >= KL+KU+1.
- **D (output)**
  The diagonal elements of the bidiagonal matrix B.
- **E (output)**

The superdiagonal elements of the bidiagonal matrix B.

- **Q (output)**
  If VECT = 'Q' or 'B', the m-by-m orthogonal matrix Q. If VECT = 'N' or 'P', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,M)` if VECT = 'Q' or 'B'; LDQ >= 1 otherwise.
- **PT (output)**
  If VECT = 'P' or 'B', the n-by-n orthogonal matrix P'. If VECT = 'N' or 'Q', the array PT is not referenced.
- **LDPT (input)**
  The leading dimension of the array PT. LDPT >= `max(1,N)` if VECT = 'P' or 'B'; LDPT >= 1 otherwise.
- **C (input/output)**
  On entry, an m-by-ncc matrix C. On exit, C is overwritten by Q'*C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)` if NCC > 0; LDC >= 1 if NCC = 0.
- **WORK (workspace)**
  `dimension(MAX(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

sgbcon - estimate the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm,

---

SYNOPSIS

```
 SUBROUTINE SGBCON( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, NSUB, NSUPER, LDA, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE SGBCON_64( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, NSUB, NSUPER, LDA, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBCON( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT, ANORM,
*        RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, NSUB, NSUPER, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GBCON_64( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT,
*        ANORM, RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, NSUB, NSUPER, LDA, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
```

```
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgbcon(char norm, int n, int nsub, int nsuper, float *a, int lda, int *ipivot, float anorm, float *rcond, int *info);

void sgbcon_64(char norm, long n, long nsub, long nsuper, float *a, long lda, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

sgbcon estimates the reciprocal of the condition number of a real general band matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**

The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  dimension (N)
- **INFO (output)**

  `= 0:  successful exit`

  `< 0: if INFO  = -i, the i-th argument had an illegal value`

NAME

sgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE SGBEQU( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC, ROWCN,
*       COLCN, AMAX, INFO)
 INTEGER M, N, NSUB, NSUPER, LDA, INFO
 REAL ROWCN, COLCN, AMAX
 REAL A(LDA,*), ROWSC(*), COLSC(*)

 SUBROUTINE SGBEQU_64( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC,
*       ROWCN, COLCN, AMAX, INFO)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
 REAL ROWCN, COLCN, AMAX
 REAL A(LDA,*), ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GBEQU( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCN, COLCN, AMAX, [INFO])
 INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
 REAL :: ROWCN, COLCN, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GBEQU_64( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*        ROWCN, COLCN, AMAX, [INFO])
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
 REAL :: ROWCN, COLCN, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgbequ(int m, int n, int nsub, int nsuper, float *a, int lda, float *rowsc, float *colsc, float *rowcn, float *colcn, float *amax, int *info);

void sgbequ_64(long m, long n, long nsub, long nsuper, float *a, long lda, float *rowsc, float *colsc, float *rowcn, float *colcn, float *amax, long *info);

PURPOSE

sgbequ computes row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input)**
  The band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **ROWSC (output)**
  If INFO = 0, or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, and i is

  < = M: the i-th row of A is exactly zero

  > M: the (i-M)-th column of A is exactly zero

NAME

sgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE SGBMV( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 INTEGER M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)

 SUBROUTINE SGBMV_64( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE GBMV( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GBMV_64( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgbmv(char transa, int m, int n, int nsub, int nsuper, float alpha, float *a, int lda, float *x, int incx, float beta, float *y, int incy);

void sgbmv_64(char transa, long m, long n, long nsub, long nsuper, float alpha, float *a, long lda, float *x, long incx, float beta, float *y, long incy);

PURPOSE

sgbmv performs one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with nsub sub-diagonals and nsuper super-diagonals.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*A'*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **NSUB (input)**
  On entry, NSUB specifies the number of sub-diagonals of the matrix A. NSUB > = 0. Unchanged on exit.
- **NSUPER (input)**
  On entry, NSUPER specifies the number of super-diagonals of the matrix A. NSUPER > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading (nsub + nsuper + 1) by n part of the array A must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (nsuper + 1) of the array, the first super-diagonal starting at position 2 in row nsuper, the first sub-diagonal starting at position 1 in row (nsuper + 2), and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left nsuper by nsuper triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
  K = NSUPER + 1 - J
  DO 10, I = MAX( 1, J - NSUPER ), MIN( M, J  + NSUB )
    A( K + I, J ) = matrix( I, J )
```

```
10   CONTINUE
20 CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (nsub + nsuper + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

sgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SGBRFS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF, LDAF,
*       IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGBRFS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*       LDAF, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBRFS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GBRFS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
```

```
REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sgbrfs(char transa, int n, int nsub, int nsuper, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void sgbrfs_64(char transa, long n, long nsub, long nsuper, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

sgbrfs improves the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` < =i < =min(n,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.
- **AF (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB*NSUPER+1.

- **IPIVOT (input)**
  The pivot indices from SGBTRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGBTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sgbsv - compute the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE SGBSV( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*       INFO)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE SGBSV_64( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*       INFO)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GBSV( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*        [LDB], [INFO])
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GBSV_64( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*        [LDB], [INFO])
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgbsv(int n, int nsub, int nsuper, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void sgbsv_64(long n, long nsub, long nsuper, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sgbsv computes the solution to a real system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = L * U, where L is a product of permutation and unit lower triangular matrices with KL subdiagonals, and U is upper triangular with KL+KU superdiagonals. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUB+NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+NSUB) On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

```
 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +       *    *    *   u14  u25  u36
 *    *    +    +    +    +       *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *      m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *      m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

sgbsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE SGBSVX( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*      LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGBSVX_64( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA,
*      AF, LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND,
*      FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSVX( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GBSVX_64( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A,
*       [LDA], AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sgbsvx(char fact, char transa, int n, int nsub, int nsuper, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, char equed, float *rowsc, float *colsc, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sgbsvx_64(char fact, char transa, long n, long nsub, long nsuper, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, char equed, float *rowsc, float *colsc, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sgbsvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed by this subroutine:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = L * U,

where L is a product of permutation and unit lower triangular
matrices with KL subdiagonals, and U is upper triangular with
KL+KU superdiagonals.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Transpose)
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.

- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the matrix A in band storage, in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+kl)

  If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).

- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns details of the LU factorization of A.

  If FACT = 'E', then AF is an output argument and on exit returns details of the LU factorization of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB+NSUPER+1.

- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = L*U as computed by SGBTRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the equilibrated matrix A.

- **EQUED (input)**

Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)` On exit, `WORK(1)` contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If `WORK(1)` is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then `WORK(1)` contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
```

```
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

sgbtf2 - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE SGBTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO)
INTEGER M, N, KL, KU, LDAB, INFO
INTEGER IPIV(*)
REAL AB(LDAB,*)

SUBROUTINE SGBTF2_64( M, N, KL, KU, AB, LDAB, IPIV, INFO)
INTEGER*8 M, N, KL, KU, LDAB, INFO
INTEGER*8 IPIV(*)
REAL AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE GBTF2( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
INTEGER :: M, N, KL, KU, LDAB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: AB

SUBROUTINE GBTF2_64( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
INTEGER(8) :: M, N, KL, KU, LDAB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void sgbtf2(int m, int n, int kl, int ku, float *ab, int ldab, int *ipiv, int *info);

void sgbtf2_64(long m, long n, long kl, long ku, float *ab, long ldab, long *ipiv, long *info);

PURPOSE

sgbtf2 computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **KL (input)**
  The number of subdiagonals within the band of A. KL >= 0.
- **KU (input)**
  The number of superdiagonals within the band of A. KU >= 0.
- **AB (input/output)**
  On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1. See below for further details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= 2*KL+KU+1.
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, KL = 2, KU = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U, because of fill-in resulting from the row

interchanges.

NAME

sgbtrf - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE SGBTRF( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
INTEGER M, N, NSUB, NSUPER, LDA, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*)

SUBROUTINE SGBTRF_64( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GBTRF( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A

SUBROUTINE GBTRF_64( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgbtrf(int m, int n, int nsub, int nsuper, float *a, int lda, int *ipivot, int *info);

void sgbtrf_64(long m, long n, long nsub, long nsuper, float *a, long lda, long *ipivot, long *info);

PURPOSE

sgbtrf computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

sgbtrs - solve a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF

SYNOPSIS

```
 SUBROUTINE SGBTRS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B,
*       LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE SGBTRS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT,
*       B, LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GBTRS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*        IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GBTRS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*        IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgbtrs(char transa, int n, int nsub, int nsuper, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void sgbtrs_64(char transa, long n, long nsub, long nsuper, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sgbtrs solves a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by SGBTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by SGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

< 0: if INFO = -i, the i-th argument had an illegal value

NAME

sgebak - form the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL

SYNOPSIS

```
 SUBROUTINE SGEBAK( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER N, ILO, IHI, M, LDV, INFO
 REAL SCALE(*), V(LDV,*)

 SUBROUTINE SGEBAK_64( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV,
*      INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 REAL SCALE(*), V(LDV,*)
```

F95 INTERFACE

```
 SUBROUTINE GEBAK( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: SCALE
 REAL, DIMENSION(:,:) :: V

 SUBROUTINE GEBAK_64( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: SCALE
 REAL, DIMENSION(:,:) :: V
```

C INTERFACE

#include <sunperf.h>

void sgebak(char job, char side, int n, int ilo, int ihi, float *scale, int m, float *v, int ldv, int *info);

void sgebak_64(char job, char side, long n, long ilo, long ihi, float *scale, long m, float *v, long ldv, long *info);

PURPOSE

sgebak forms the right or left eigenvectors of a real general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by SGEBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required: = 'N', do nothing, return immediately; = 'P', do backward transformation for permutation only; = 'S', do backward transformation for scaling only; = 'B', do backward transformations for both permutation and scaling. JOB must be the same as the argument JOB supplied to SGEBAL.
- **SIDE (input)**

  ```
  = 'R':  V contains right eigenvectors;

  = 'L':  V contains left eigenvectors.
  ```
- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by SGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description for ILO.
- **SCALE (input)**
  Details of the permutation and scaling factors, as returned by SGEBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**
  On entry, the matrix of right or left eigenvectors to be transformed, as returned by SHSEIN or STREVC. On exit, V is overwritten by the transformed eigenvectors.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

sgebal - balance a general real matrix A

SYNOPSIS

```
SUBROUTINE SGEBAL( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
INTEGER N, LDA, ILO, IHI, INFO
REAL A(LDA,*), SCALE(*)

SUBROUTINE SGEBAL_64( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
INTEGER*8 N, LDA, ILO, IHI, INFO
REAL A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE GEBAL( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER :: N, LDA, ILO, IHI, INFO
REAL, DIMENSION(:) :: SCALE
REAL, DIMENSION(:,:) :: A

SUBROUTINE GEBAL_64( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
INTEGER(8) :: N, LDA, ILO, IHI, INFO
REAL, DIMENSION(:) :: SCALE
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgebal(char job, int n, float *a, int lda, int *ilo, int *ihi, float *scale, int *info);

void sgebal_64(char job, long n, float *a, long lda, long *ilo, long *ihi, float *scale, long *info);

PURPOSE

sgebal balances a general real matrix A. This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to ILO-1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, SCALE(I)  = 1.0
for i  = 1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description for ILO.
- **SCALE (output)**
  Details of the permutations and scaling factors applied to A. If `P(j)` is the index of the row and column interchanged with row and column j and `D(j)` is the scaling factor applied to row and column j, then `SCALE(j)` = `P(j)` for j = 1,...,ILO-1 = `D(j)` for j = ILO,...,IHI = `P(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The permutations consist of row and column interchanges which put the matrix in the form

```
           ( T1   X   Y  )
P A P  = (  0   B   Z  )
           (  0   0   T2 )
```

where T1 and T2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ILO and IHI mark the starting and ending columns of the submatrix B. Balancing consists of applying a diagonal similarity transformation `inv(D)` * B * D to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

```
( T1     X*D          Y    )
(  0  inv(D)*B*D  inv(D)*Z ).
(  0      0           T2   )
```

Information about the permutations P and the diagonal matrix D is returned in the vector SCALE.

This subroutine is based on the EISPACK routine BALANC.

Modified by Tzu-Yi Chen, Computer Science Division, University of California at Berkeley, USA

NAME

sgebrd - reduce a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation

---

SYNOPSIS

```
 SUBROUTINE SGEBRD( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 INTEGER M, N, LDA, LWORK, INFO
 REAL A(LDA,*), D(*), E(*), TAUQ(*), TAUP(*), WORK(*)

 SUBROUTINE SGEBRD_64( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 INTEGER*8 M, N, LDA, LWORK, INFO
 REAL A(LDA,*), D(*), E(*), TAUQ(*), TAUP(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBRD( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 INTEGER :: M, N, LDA, LWORK, INFO
 REAL, DIMENSION(:) :: D, E, TAUQ, TAUP, WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GEBRD_64( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 REAL, DIMENSION(:) :: D, E, TAUQ, TAUP, WORK
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgebrd(int m, int n, float *a, int lda, float *d, float *e, float *tauq, float *taup, int *info);

void sgebrd_64(long m, long n, float *a, long lda, float *d, float *e, float *tauq, float *taup, long *info);

PURPOSE

sgebrd reduces a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation: Q**T * A * P = B.

If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.

ARGUMENTS

- **M (input)**
  The number of rows in the matrix A. M > = 0.
- **N (input)**
  The number of columns in the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N general matrix to be reduced. On exit, if m > = n, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors; if m < n, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **D (output)**
  The diagonal elements of the bidiagonal matrix B: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the bidiagonal matrix B: if m > = n, `E(i)` = `A(i,i+1)` for i = 1,2,...,n-1; if m < n, `E(i)` = `A(i+1,i)` for i = 1,2,...,m-1.
- **TAUQ (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.
- **TAUP (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,M,N). For optimum performance LWORK > = (M+N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrices Q and P are represented as products of elementary reflectors:

If m >= n,

```
Q  = H(1) H(2) . . . H(n)  and  P  = G(1) G(2) . . . G(n-1)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are real scalars, and v and u are real vectors; `v(1:i-1)` = 0, `v(i)` = 1, and `v(i+1:m)` is stored on exit in A(i+1:m,i); `u(1:i)` = 0, `u(i+1)` = 1, and `u(i+2:n)` is stored on exit in A(i,i+2:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

If m < n,

```
Q  = H(1) H(2) . . . H(m-1)  and  P  = G(1) G(2) . . . G(m)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are real scalars, and v and u are real vectors; `v(1:i)` = 0, `v(i+1)` = 1, and `v(i+2:m)` is stored on exit in A(i+2:m,i); `u(1:i-1)` = 0, `u(i)` = 1, and `u(i+1:n)` is stored on exit in A(i,i+1:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

The contents of A on exit are illustrated by the following examples:

m = 6 and n = 5 (m > n): m = 5 and n = 6 (m < n):

```
(  d   e   u1  u1  u1 )          (  d   u1  u1  u1  u1  u1 )
(  v1  d   e   u2  u2 )          (  e   d   u2  u2  u2  u2 )
(  v1  v2  d   e   u3 )          (  v1  e   d   u3  u3  u3 )
(  v1  v2  v3  d   e  )          (  v1  v2  e   d   u4  u4 )
(  v1  v2  v3  v4  d  )          (  v1  v2  v3  e   d   u5 )
(  v1  v2  v3  v4  v5 )
```

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining H(i), and ui an element of the vector defining G(i).

NAME

sgecon - estimate the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF

SYNOPSIS

```
 SUBROUTINE SGECON( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, LDA, INFO
 INTEGER WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE SGECON_64( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, LDA, INFO
 INTEGER*8 WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GECON( NORM, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GECON_64( NORM, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL :: ANORM, RCOND
```

```
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgecon(char norm, int n, float *a, int lda, float anorm, float *rcond, int *info);

void sgecon_64(char norm, long n, float *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

sgecon estimates the reciprocal of the condition number of a general real matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(4*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE SGEEQU( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN, AMAX,
*       INFO)
 INTEGER M, N, LDA, INFO
 REAL ROWCN, COLCN, AMAX
 REAL A(LDA,*), ROWSC(*), COLSC(*)

 SUBROUTINE SGEEQU_64( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN,
*       AMAX, INFO)
 INTEGER*8 M, N, LDA, INFO
 REAL ROWCN, COLCN, AMAX
 REAL A(LDA,*), ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEQU( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*        AMAX, [INFO])
 INTEGER :: M, N, LDA, INFO
 REAL :: ROWCN, COLCN, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GEEQU_64( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*        AMAX, [INFO])
 INTEGER(8) :: M, N, LDA, INFO
 REAL :: ROWCN, COLCN, AMAX
 REAL, DIMENSION(:) :: ROWSC, COLSC
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgeequ(int m, int n, float *a, int lda, float *rowsc, float *colsc, float *rowcn, float *colcn, float *amax, int *info);

void sgeequ_64(long m, long n, float *a, long lda, float *rowsc, float *colsc, float *rowcn, float *colcn, float *amax, long *info);

PURPOSE

sgeequ computes row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input)**
  The M-by-N matrix whose equilibration factors are to be computed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **ROWSC (output)**
  If INFO = 0 or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: if INFO = i, and i is

< = M: the i-th row of A is exactly zero

> M: the (i-M)-th column of A is exactly zero

NAME

sgees - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE SGEES( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, WR, WI, Z,
*      LDZ, WORK, LDWORK, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL WORK3(*)
 REAL A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SGEES_64( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, WR, WI,
*      Z, LDZ, WORK, LDWORK, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 WORK3(*)
 REAL A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEES( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, WR, WI,
*       Z, [LDZ], [WORK], [LDWORK], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: WORK3
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: A, Z

 SUBROUTINE GEES_64( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, WR,
*       WI, Z, [LDZ], [WORK], [LDWORK], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL(8) :: SELECT
```

```
LOGICAL(8), DIMENSION(:) :: WORK3
REAL, DIMENSION(:) :: WR, WI, WORK
REAL, DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void sgees(char jobz, char sortev, logical(*select)(float,float), int n, float *a, int lda, int *nout, float *wr, float *wi, float *z, int ldz, int *info);

void sgees_64(char jobz, char sortev, logical(*select)(float,float), long n, float *a, long lda, long *nout, float *wr, float *wi, float *z, long ldz, long *info);

PURPOSE

sgees computes for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**T).

Optionally, it also orders the eigenvalues on the diagonal of the real Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A matrix is in real Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

```
[  a  b  ]

[  c  a  ]
```

where b*c < 0. The eigenvalues of such a block are a +- sqrt(bc).

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.
  ```

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

  ```
  = 'S': Eigenvalues are ordered (see SELECT).
  ```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to sort to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `WR(j)+sqrt(-1)*WI(j)` is selected if `SELECT(WR(j),WI(j))` is true; i.e., if either one of a complex

conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy SELECT(WR(j),WI(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case INFO is set to N+2 (see INFO below).

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten by its real Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues (after sorting) for which SELECT is true. (Complex conjugate pairs for which SELECT is true for either eigenvalue count as 2.)
- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues in the same order that they appear on the diagonal of the output Schur form T. Complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **Z (output)**
  If JOBZ = 'V', Z contains the orthogonal matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1; if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) contains the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,3*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK3 (workspace)**
  dimension(N) Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N: the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of WR and WI contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the matrix which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

sgeesx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE SGEESX( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT, WR,
*      WI, Z, LDZ, SRCONE, RCONV, WORK, LDWORK, IWORK2, LDWRK2, BWORK3,
*      INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 INTEGER N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER IWORK2(*)
 LOGICAL SELECT
 LOGICAL BWORK3(*)
 REAL SRCONE, RCONV
 REAL A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SGEESX_64( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT,
*      WR, WI, Z, LDZ, SRCONE, RCONV, WORK, LDWORK, IWORK2, LDWRK2,
*      BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER*8 IWORK2(*)
 LOGICAL*8 SELECT
 LOGICAL*8 BWORK3(*)
 REAL SRCONE, RCONV
 REAL A(LDA,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEESX( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA], NOUT,
*       WR, WI, Z, [LDZ], SRCONE, RCONV, [WORK], [LDWORK], [IWORK2],
*       [LDWRK2], [BWORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
 INTEGER, DIMENSION(:) :: IWORK2
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: BWORK3
```

```
  REAL :: SRCONE, RCONV
  REAL, DIMENSION(:) :: WR, WI, WORK
  REAL, DIMENSION(:,:) :: A, Z

  SUBROUTINE GEESX_64( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA],
*       NOUT, WR, WI, Z, [LDZ], SRCONE, RCONV, [WORK], [LDWORK], [IWORK2],
*       [LDWRK2], [BWORK3], [INFO])
  CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
  INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, LDWRK2, INFO
  INTEGER(8), DIMENSION(:) :: IWORK2
  LOGICAL(8) :: SELECT
  LOGICAL(8), DIMENSION(:) :: BWORK3
  REAL :: SRCONE, RCONV
  REAL, DIMENSION(:) :: WR, WI, WORK
  REAL, DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void sgeesx(char jobz, char sortev, logical(*select)(float,float), char sense, int n, float *a, int lda, int *nout, float *wr, float *wi, float *z, int ldz, float *srcone, float *rconv, int *info);

void sgeesx_64(char jobz, char sortev, logical(*select)(float,float), char sense, long n, float *a, long lda, long *nout, float *wr, float *wi, float *z, long ldz, float *srcone, float *rconv, long *info);

PURPOSE

sgeesx computes for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**T).

Optionally, it also orders the eigenvalues on the diagonal of the real Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (RCONDV). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see Section 4.10 of the LAPACK Users' Guide (where these quantities are called s and sep respectively).

A real matrix is in real Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

```
          [  a  b  ]

          [  c  a  ]
```

where b*c < 0. The eigenvalues of such a block are a +- sqrt(bc).

ARGUMENTS

- **JOBZ (input)**

```
= 'N': Schur vectors are not computed;

= 'V': Schur vectors are computed.
```

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S': Eigenvalues are ordered (see SELECT).
```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to sort to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `WR(j)+sqrt(-1)*WI(j)` is selected if `SELECT(WR(j),WI(j))` is true; i.e., if either one of a complex conjugate pair of eigenvalues is selected, then both are. Note that a selected complex eigenvalue may no longer satisfy `SELECT(WR(j),WI(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case INFO may be set to N+3 (see INFO below).
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
= 'E': Computed for average of selected eigenvalues only;

= 'V': Computed for selected right invariant subspace only;

= 'B': Computed for both.
If SENSE  = 'E', 'V' or 'B', SORTEV must equal 'S'.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A is overwritten by its real Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues (after sorting) for which SELECT is true. (Complex conjugate pairs for which SELECT is true for either eigenvalue count as 2.)
- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.
- **WI (output)**
  See the description for WR.
- **Z (output)**
  If JOBZ = 'V', Z contains the orthogonal matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.
- **SRCONE (output)**
  If SENSE = 'E' or 'B', SRCONE contains the reciprocal condition number for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONV (output)**

If SENSE = 'V' or 'B', RCONV contains the reciprocal condition number for the selected right invariant subspace. Not referenced if SENSE = 'N' or 'E'.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,3*N). Also, if SENSE = 'E' or 'V' or 'B', LDWORK > = N+2*NOUT*(N-NOUT), where NOUT is the number of selected eigenvalues computed by this routine. Note that N+2*NOUT*(N-NOUT) < = N+N*N/2. For good performance, LDWORK must generally be larger.
- **IWORK2 (workspace)**
  Not referenced if SENSE = 'N' or 'E'. On exit, if INFO = 0, `IWORK2(1)` returns the optimal LDWRK2.
- **LDWRK2 (input)**
  The dimension of the array IWORK2. LDWRK2 > = 1; if SENSE = 'V' or 'B', LDWRK2 > = NOUT*(N-NOUT).
- **BWORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N: the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of WR and WI contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the transformation which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

sgeev - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE SGEEV( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR,
*       LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE SGEEV_64( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR,
*       LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER*8 N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEV( JOBVL, JOBVR, [N], A, [LDA], WR, WI, VL, [LDVL],
*        VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: A, VL, VR

 SUBROUTINE GEEV_64( JOBVL, JOBVR, [N], A, [LDA], WR, WI, VL, [LDVL],
*        VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER(8) :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: A, VL, VR
```

C INTERFACE

#include <sunperf.h>

void sgeev(char jobvl, char jobvr, int n, float *a, int lda, float *wr, float *wi, float *vl, int ldvl, float *vr, int ldvr, int *info);

void sgeev_64(char jobvl, char jobvr, long n, float *a, long lda, float *wr, float *wi, float *vl, long ldvl, float *vr, long ldvr, long *info);

PURPOSE

sgeev computes for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N': left eigenvectors of A are not computed;

  = 'V': left eigenvectors of A are computed.
  ```

- **JOBVR (input)**

  ```
  = 'N': right eigenvectors of A are not computed;

  = 'V': right eigenvectors of A are computed.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **WR (output)**

WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.

- **WI (output)**
  See the description for WR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)` + i*VL(:,j+1) and

  `u(j+1)` = `VL(:,j)` - i*VL(:,j+1).
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)` + i*VR(:,j+1) and

  `v(j+1)` = `VR(:,j)` - i*VR(:,j+1).
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,3*N), and if JOBVL = 'V' or JOBVR = 'V', LDWORK >= 4*N. For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors have been computed;
elements i+1:N of WR and WI contain eigenvalues which
have converged.
```

NAME

sgeevx - compute for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE SGEEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, WR, WI,
*      VL, LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, IWORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 INTEGER N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER IWORK2(*)
 REAL ABNRM
 REAL A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), SCALE(*), RCONE(*), RCONV(*)
 DOUBLE PRECISION WORK(*)

 SUBROUTINE SGEEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, WR,
*      WI, VL, LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*      WORK, LDWORK, IWORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 INTEGER*8 N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER*8 IWORK2(*)
 REAL ABNRM
 REAL A(LDA,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), SCALE(*), RCONE(*), RCONV(*)
 DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], WR,
*       WI, VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*       WORK, [LDWORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 INTEGER :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK2
 REAL :: ABNRM
 REAL, DIMENSION(:) :: WR, WI, SCALE, RCONE, RCONV
 REAL(8), DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, VL, VR
```

```
 SUBROUTINE GEEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], WR,
*       WI, VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
*       WORK, [LDWORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 INTEGER(8) :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2
 REAL :: ABNRM
 REAL, DIMENSION(:) :: WR, WI, SCALE, RCONE, RCONV
 REAL(8), DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, VL, VR
```

C INTERFACE

#include <sunperf.h>

void sgeevx(char balanc, char jobvl, char jobvr, char sense, int n, float *a, int lda, float *wr, float *wi, float *vl, int ldvl, float *vr, int ldvr, int *ilo, int *ihi, float *scale, float *abnrm, float *rcone, float *rconv, double *work, int ldwork, int *info);

void sgeevx_64(char balanc, char jobvl, char jobvr, char sense, long n, float *a, long lda, float *wr, float *wi, float *vl, long ldvl, float *vr, long ldvr, long *ilo, long *ihi, float *scale, float *abnrm, float *rcone, float *rconv, double *work, long ldwork, long *info);

PURPOSE

sgeevx computes for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, SCALE, and ABNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right

eigenvectors (RCONDV).

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D**(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide.

ARGUMENTS

- **BALANC (input)**
  Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. = 'N': Do not diagonally scale or permute;

  ```
   = 'P': Perform permutations to make the matrix more nearly
  upper triangular. Do not diagonally scale;
   = 'S': Diagonally scale the matrix, i.e. replace A by
  D*A*D**(-1), where D is a diagonal matrix chosen
  to make the rows and columns of A more equal in
  norm. Do not permute;
   = 'B': Both diagonally scale and permute A.
  ```

  Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

- **JOBVL (input)**

  ```
   = 'N': left eigenvectors of A are not computed;

   = 'V': left eigenvectors of A are computed.
  If SENSE  = 'E' or 'B', JOBVL must  = 'V'.
  ```

- **JOBVR (input)**

  ```
   = 'N': right eigenvectors of A are not computed;

   = 'V': right eigenvectors of A are computed.
  If SENSE  = 'E' or 'B', JOBVR must  = 'V'.
  ```

- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

  ```
   = 'E': Computed for eigenvalues only;

   = 'V': Computed for right eigenvectors only;

   = 'B': Computed for eigenvalues and right eigenvectors.
  ```

  If SENSE = 'E' or 'B', both left and right eigenvectors must also be computed (JOBVL = 'V' and JOBVR = 'V').

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten. If JOBVL = 'V' or JOBVR = 'V', A contains the real Schur form of the balanced version of the input matrix A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **WR (output)**
  WR and WI contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.

- **WI (output)**
  See the description for WR.

- **VL (output)**

If JOBVL = 'V', the left eigenvectors u(j) are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. If the j-th eigenvalue is real, then u(j) = VL(:,j), the j-th column of VL. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then u(j) = VL(:,j) + i*VL(:,j+1) and

u(j+1) = VL(:,j) - i*VL(:,j+1).

- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors v(j) are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. If the j-th eigenvalue is real, then v(j) = VR(:,j), the j-th column of VR. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then v(j) = VR(:,j) + i*VR(:,j+1) and

  v(j+1) = VR(:,j) - i*VR(:,j+1).
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced A(i,j) = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **IHI (output)**
  See the description of ILO.
- **SCALE (output)**
  Details of the permutations and scaling factors applied when balancing A. If P(j) is the index of the row and column interchanged with row and column j, and D(j) is the scaling factor applied to row and column j, then SCALE(J) = P(J), for J = 1,...,ILO-1 = D(J), for J = ILO,...,IHI = P(J) for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
- **RCONE (output)**
  RCONE(j) is the reciprocal condition number of the j-th eigenvalue.
- **RCONV (output)**
  RCONV(j) is the reciprocal condition number of the j-th right eigenvector.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. If SENSE = 'N' or 'E', LDWORK >= max(1,2*N), and if JOBVL = 'V' or JOBVR = 'V', LDWORK >= 3*N. If SENSE = 'V' or 'B', LDWORK >= N*(N+6). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **IWORK2 (workspace)**
  dimension(2*N-2) If SENSE = 'N' or 'E', not referenced.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.

> 0:  if INFO  = i, the QR algorithm failed to compute all the
```

eigenvalues, and no eigenvectors or condition numbers
have been computed; elements 1:ILO-1 and i+1:N of WR
and WI contain eigenvalues which have converged.

NAME

sgegs - routine is deprecated and has been replaced by routine SGGES

SYNOPSIS

```
 SUBROUTINE SGEGS( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
*      BETA, VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 INTEGER N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)

 SUBROUTINE SGEGS_64( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHAR,
*      ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 INTEGER*8 N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGS( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 INTEGER :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR

 SUBROUTINE GEGS_64( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 INTEGER(8) :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void sgegs(char jobvsl, char jobvsr, int n, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *vsl, int ldvsl, float *vsr, int ldvsr, int *info);

void sgegs_64(char jobvsl, char jobvsr, long n, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *vsl, long ldvsl, float *vsr, long ldvsr, long *info);

PURPOSE

sgegs routine is deprecated and has been replaced by routine SGGES.

SGEGS computes for a pair of N-by-N real nonsymmetric matrices A, B: the generalized eigenvalues (alphar +/- alphai*i, beta), the real Schur form (A, B), and optionally left and/or right Schur vectors (VSL and VSR).

(If only the generalized eigenvalues are needed, use the driver SGEGV instead.)

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations'', by G. Golub & C. van Loan (Johns Hopkins U. Press)

The (generalized) Schur form of a pair of matrices is the result of multiplying both matrices on the left by one orthogonal matrix and both on the right by another orthogonal matrix, these two orthogonal matrices being chosen so as to bring the pair of matrices into (real) Schur form.

A pair of matrices A, B is in generalized real Schur form if B is upper triangular with non-negative diagonal and A is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of A will be ``standardized'' by making the corresponding elements of B have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in A and B will have a complex conjugate pair of generalized eigenvalues.

The left and right Schur vectors are the columns of VSL and VSR, respectively, where VSL and VSR are the orthogonal matrices which reduce A and B to Schur form:

Schur form of (A,B) = ((VSL)**T A (VSR), (VSL)**T B (VSR))

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```
- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```
- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of A. Note: to avoid overflow, the Frobenius norm of the matrix A should be less than the overflow threshold.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On

exit, the generalized Schur form of B. Note: to avoid overflow, the Frobenius norm of the matrix B should be less than the overflow threshold.

- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHAR(j) + ALPHAI(j)*i, j =1,...,N and BETA(j),j =1,...,N are the diagonals of the complex Schur form (A,B) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If ALPHAI(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with ALPHAI(j+1) negative.

  Note: the quotients ALPHAR(j)/BETA(j) and ALPHAI(j)/BETA(j) may easily over- or underflow, and BETA(j) may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with norm(A) in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > =1, and if JOBVSL = 'V', LDVSL > = N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR > = 1, and if JOBVSR = 'V', LDVSR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,4*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for SGEQRF, SORMQR, and SORGQR.) Then compute: NB -- MAX of the blocksizes for SGEQRF, SORMQR, and SORGQR The optimal LDWORK is 2*N + N*(NB+1).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from SGGBAL

 =N+2: error return from SGEQRF

 =N+3: error return from SORMQR

 =N+4: error return from SORGQR

 =N+5: error return from SGGHRD
```

```
 =N+6: error return from SHGEQZ (other than failed
iteration)
 =N+7: error return from SGGBAK (computing VSL)

 =N+8: error return from SGGBAK (computing VSR)

 =N+9: error return from SLASCL (various places)
```

NAME

sgegv - routine is deprecated and has been replaced by routine SGGEV

SYNOPSIS

```
 SUBROUTINE SGEGV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
*       BETA, VL, LDVL, VR, LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE SGEGV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR,
*       ALPHAI, BETA, VL, LDVL, VR, LDVR, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE GEGV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void sgegv(char jobvl, char jobvr, int n, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *vl, int ldvl, float *vr, int ldvr, int *info);

void sgegv_64(char jobvl, char jobvr, long n, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *vl, long ldvl, float *vr, long ldvr, long *info);

PURPOSE

sgegv routine is deprecated and has been replaced by routine SGGEV.

SGEGV computes for a pair of n-by-n real nonsymmetric matrices A and B, the generalized eigenvalues (alphar +/- alphai*i, beta), and optionally, the left and/or right generalized eigenvectors (VL and VR).

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations'', by G. Golub & C. van Loan (Johns Hopkins U. Press)

A right generalized eigenvector corresponding to a generalized eigenvalue w for a pair of matrices (A,B) is a vector r such that (A - w B) r = 0 . A left generalized eigenvector is a vector l such that l**H * (A - w B) = 0, where l**H is the

conjugate-transpose of l.

Note: this routine performs ``full balancing'' on A and B -- see ``Further Details'', below.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of A on exit, see ``Further Details'', below.)

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of B on exit, see ``Further

Details", below.)

- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose", above.) Real eigenvectors take one column, complex take two columns, the first for the real part and the second for the imaginary part. Complex eigenvectors correspond to an eigenvalue with positive imaginary part. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors. (See ``Purpose", above.) Real eigenvectors take one column, complex take two columns, the first for the real part and the second for the imaginary part. Complex eigenvectors correspond to an eigenvalue with positive imaginary part. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,8*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for SGEQRF, SORMQR, and SORGQR.) Then compute: NB -- MAX of the blocksizes for SGEQRF, SORMQR, and SORGQR; The optimal LDWORK is: 2*N + MAX(6*N, N*(NB+1)).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from SGGBAL

 =N+2: error return from SGEQRF
```

```
 =N+3: error return from SORMQR

 =N+4: error return from SORGQR

 =N+5: error return from SGGHRD

 =N+6: error return from SHGEQZ (other than failed
iteration)
 =N+7: error return from STGEVC

 =N+8: error return from SGGBAK (computing VL)

 =N+9: error return from SGGBAK (computing VR)

 =N+10: error return from SLASCL (various calls)
```

FURTHER DETAILS

Balancing

This driver calls SGGBAL to both permute and scale rows and columns of A and B. The permutations PL and PR are chosen so that PL*A*PR and PL*B*R will be upper triangular except for the diagonal blocks `A(i:j,i:j)` and B(i:j,i:j), with i and j as close together as possible. The diagonal scaling matrices DL and DR are chosen so that the pair DL*PL*A*PR*DR, DL*PL*B*PR*DR have elements close to one (except for the elements that start out zero.)

After the eigenvalues and eigenvectors of the balanced matrices have been computed, SGGBAK transforms the eigenvectors back to what they would have been (in perfect arithmetic) if they had not been balanced.

Contents of A and B on Exit

-------- -- - --- - -- ----

If any eigenvectors are computed (either JOBVL ='V' or JOBVR ='V' or both), then on exit the arrays A and B will contain the real Schur form[*] of the ``balanced'' versions of A and B. If no eigenvectors are computed, then only the diagonal blocks will be correct.

[*] See SHGEQZ, SGEGS, or read the book ``Matrix Computations'', by Golub & van Loan, pub. by Johns Hopkins U. Press.

NAME

sgehrd - reduce a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation

---

SYNOPSIS

```
  SUBROUTINE SGEHRD( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN, INFO)
  INTEGER N, ILO, IHI, LDA, LWORKIN, INFO
  REAL A(LDA,*), TAU(*), WORKIN(*)

  SUBROUTINE SGEHRD_64( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN,
 *      INFO)
  INTEGER*8 N, ILO, IHI, LDA, LWORKIN, INFO
  REAL A(LDA,*), TAU(*), WORKIN(*)
```

F95 INTERFACE

```
  SUBROUTINE GEHRD( [N], ILO, IHI, A, [LDA], TAU, [WORKIN], [LWORKIN],
 *       [INFO])
  INTEGER :: N, ILO, IHI, LDA, LWORKIN, INFO
  REAL, DIMENSION(:) :: TAU, WORKIN
  REAL, DIMENSION(:,:) :: A

  SUBROUTINE GEHRD_64( [N], ILO, IHI, A, [LDA], TAU, [WORKIN],
 *       [LWORKIN], [INFO])
  INTEGER(8) :: N, ILO, IHI, LDA, LWORKIN, INFO
  REAL, DIMENSION(:) :: TAU, WORKIN
  REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgehrd(int n, int ilo, int ihi, float *a, int lda, float *tau, int *info);

void sgehrd_64(long n, long ilo, long ihi, float *a, long lda, float *tau, long *info);

PURPOSE

sgehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation: Q' * A * Q = H .

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGEBAL; otherwise they should be set to 1 and N respectively. See Further Details.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details). Elements 1:ILO-1 and IHI:N-1 of TAU are set to zero.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The length of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of (ihi-ilo) elementary reflectors

```
Q  = H(ilo) H(ilo+1) . . . H(ihi-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0, `v(i+1)` = 1 and `v(ihi+1:n)` = 0; `v(i+2:ihi)` is stored on exit in A(i+2:ihi,i), and tau in TAU(i).

The contents of A are illustrated by the following example, with n = 7, ilo = 2 and ihi = 6:

on entry, on exit,

(a a a a a a a) (a a h h h h a) (a a a a a a) (a h h h h a) (a a a a a a) (h h h h h h) (a a a a a a) (v2 h h h h h) (a a a a a a) (v2 v3 h h h h) (a a a a a a) (v2 v3 v4 h h h) (a) (a)

where a denotes an element of the original matrix A, h denotes a modified element of the upper Hessenberg matrix H, and vi denotes an element of the vector defining H(i).

NAME

sgelqf - compute an LQ factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE SGELQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGELQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GELQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GELQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgelqf(int m, int n, float *a, int lda, float *tau, int *info);

void sgelqf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

sgelqf computes an LQ factorization of a real M-by-N matrix A: A = L * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and below the diagonal of the array contain the m-by-min(m,n) lower trapezoidal matrix L (L is lower triangular if m <= n); the elements above the diagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i,i+1:n), and tau in TAU(i).

NAME

sgels - solve overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A

SYNOPSIS

```
 SUBROUTINE SGELS( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK, LDWORK,
*      INFO)
 CHARACTER * 1 TRANSA
 INTEGER M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE SGELS_64( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELS( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GELS_64( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, NRHS, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgels(char transa, int m, int n, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void sgels_64(char transa, long m, long n, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

sgels solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||.

2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B.

3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B.

4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

ARGUMENTS

- **TRANSA (input)**

  ```
  = 'N': the linear system involves A;

  = 'T': the linear system involves A**T.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > =0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M > = N, A is overwritten by details of its QR factorization as returned by SGEQRF; if M < N, A is overwritten by details of its LQ factorization as returned by SGELQF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the matrix B of right hand side vectors, stored columnwise; B is M-by-NRHS if TRANSA = 'N', or

N-by-NRHS if TRANSA = 'T'. On exit, B is overwritten by the solution vectors, stored columnwise: if TRANSA = 'N' and m >= n, rows 1 to n of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements N+1 to M in that column; if TRANSA = 'N' and m < n, rows 1 to N of B contain the minimum norm solution vectors; if TRANSA = 'T' and m >= n, rows 1 to M of B contain the minimum norm solution vectors; if TRANSA = 'T' and m < n, rows 1 to M of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements M+1 to N in that column.

- **LDB (input)**
  The leading dimension of the array B. LDB >= MAX(1,M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1, MN + max(MN, NRHS)). For optimal performance, LDWORK >= max(1, MN + max(MN, NRHS)*NB). where MN = `min(M,N)` and NB is the optimum block size.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sgelsd - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE SGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK,
*      LWORK, IWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER IWORK(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), S(*), WORK(*)

 SUBROUTINE SGELSD_64( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK,
*      WORK, LWORK, IWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), S(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSD( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [IWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: RCOND
 REAL, DIMENSION(:) :: S, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GELSD_64( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [IWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL :: RCOND
 REAL, DIMENSION(:) :: S, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgelsd(int m, int n, int nrhs, float *a, int lda, float *b, int ldb, float *s, float rcond, int *rank, int *info);

void sgelsd_64(long m, long n, long nrhs, float *a, long lda, float *b, long ldb, float *s, float rcond, long *rank, long *info);

PURPOSE

sgelsd computes the minimum-norm solution to a real linear least squares problem: minimize 2-norm(| b - A*x |)

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The problem is solved in three steps:

(1) Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a ``bidiagonal least squares problem'' (BLS)

(2) Solve the BLS using a divide and conquer approach.

(3) Apply back all the Householder tranformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **M (input)**
  The number of rows of A. M >= 0.
- **N (input)**
  The number of columns of A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m >= n and RANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,max(M,N)).
- **S (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = S(1)/S(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `S(i)` <= RCOND*S(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **RANK (output)**
  The effective rank of A, i.e., the number of singular values which are greater than RCOND*S(1).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. The exact minimum amount of workspace needed depends on M, N and NRHS. As long as LWORK is at least 12*N + 2*N*SMLSIZ + 8*N*NLVL + N*NRHS * (SMLSIZ+1)**2, if M is greater than or equal to N

or 12*M + 2*M*SMLSIZ + 8*M*NLVL + M*NRHS + (SMLSIZ+1)**2, if M is less than N, the code will execute correctly. SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and NLVL = INT(LOG_2(MIN(M,N)/(SMLSIZ+1))) + 1 For good performance, LWORK should generally be larger.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  LIWORK > = 3 * MINMN * NLVL + 11 * MINMN, where MINMN = MIN(M,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Ren-Cang Li, Computer Science Division, University of California at Berkeley, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

sgelss - compute the minimum norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE SGELSS( M, N, NRHS, A, LDA, B, LDB, SING, RCOND, IRANK,
*      WORK, LDWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), SING(*), WORK(*)

 SUBROUTINE SGELSS_64( M, N, NRHS, A, LDA, B, LDB, SING, RCOND,
*      IRANK, WORK, LDWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), SING(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSS( [M], [N], [NRHS], A, [LDA], B, [LDB], SING, RCOND,
*       IRANK, [WORK], [LDWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SING, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GELSS_64( [M], [N], [NRHS], A, [LDA], B, [LDB], SING,
*       RCOND, IRANK, [WORK], [LDWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: SING, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgelss(int m, int n, int nrhs, float *a, int lda, float *b, int ldb, float *sing, float rcond, int *irank, int *info);

void sgelss_64(long m, long n, long nrhs, float *a, long lda, float *b, long ldb, float *sing, float rcond, long *irank, long *info);

PURPOSE

sgelss computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm(| b - A*x |).

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the first `min(m,n)` rows of A are overwritten with its right singular vectors, stored rowwise.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,max(M,N)).
- **SING (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = SING(1)/SING(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `SING(i)` < = RCOND*SING(1) are treated as zero. If RCOND < 0, machine precision is used instead.
- **IRANK (output)**

The effective rank of A, i.e., the number of singular values which are greater than RCOND*SING(1).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.

- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1, and also: LDWORK >= 3*min(M,N) + max(2*min(M,N), max(M,N), NRHS) For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

NAME

sgelsx - routine is deprecated and has been replaced by routine SGELSY

SYNOPSIS

```
 SUBROUTINE SGELSX( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND, IRANK,
*      WORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER JPIVOT(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE SGELSX_64( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND,
*      IRANK, WORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER*8 JPIVOT(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSX( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER, DIMENSION(:) :: JPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GELSX_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER(8), DIMENSION(:) :: JPIVOT
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgelsx(int m, int n, int nrhs, float *a, int lda, float *b, int ldb, int *jpivot, float rcond, int *irank, int *info);

void sgelsx_64(long m, long n, long nrhs, float *a, long lda, float *b, long ldb, long *jpivot, float rcond, long *irank, long *info);

PURPOSE

sgelsx routine is deprecated and has been replaced by routine SGELSY.

SGELSX computes the minimum-norm solution to a real linear least squares problem:

```
    minimize || A * X - B ||
```

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X. If m >= n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements N+1:M in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is an initial column, otherwise it is a free column. Before the QR factorization of A, all initial columns are permuted to the leading positions; only the remaining free columns are moved as a result of column pivoting during the factorization. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **IRANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  (max(min(M,N)+3*N, 2*min(M,N)+NRHS)),
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sgelsy - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE SGELSY( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, INFO)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER JPVT(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE SGELSY_64( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, INFO)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 JPVT(*)
 REAL RCOND
 REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSY( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT, RCOND,
*       RANK, [WORK], [LWORK], [INFO])
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GELSY_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT,
*       RCOND, RANK, [WORK], [LWORK], [INFO])
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: JPVT
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgelsy(int m, int n, int nrhs, float *a, int lda, float *b, int ldb, int *jpvt, float rcond, int *rank, int *info);

void sgelsy_64(long m, long n, long nrhs, float *a, long lda, float *b, long ldb, long *jpvt, float rcond, long *rank, long *info);

PURPOSE

sgelsy computes the minimum-norm solution to a real linear least squares problem: minimize || A * X - B ||

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

This routine is basically identical to the original xGELSX except three differences:

```
  o The call to the subroutine xGEQPF has been substituted by the
    the call to the subroutine xGEQP3. This subroutine is a Blas-3
    version of the QR factorization with column pivoting.
  o Matrix B (the right hand side) is updated with Blas-3.
  o The permutation of matrix B (the right hand side) is faster and
    more simple.
```

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPVT (input/output)**
  On entry, if `JPVT(i)` .ne. 0, the i-th column of A is permuted to the front of AP, otherwise column i is a free column. On exit, if `JPVT(i)` = k, then the i-th column of AP was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **RANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. The unblocked strategy requires that: LWORK >= MAX(MN+3*N+1, 2*MN+NRHS), where MN = min(M, N). The block algorithm requires that: LWORK >= MAX(MN+2*N+NB*(N+1), 2*MN+NB*NRHS), where NB is an upper bound on the blocksize returned by ILAENV for the routines SGEQP3, STZRZF, STZRQF, SORMQR, and SORMRZ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0: successful exit

< 0: If INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
E. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain

---

NAME

sgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

SYNOPSIS

```
 SUBROUTINE SGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 INTEGER M, N, K, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE SGEMM_64( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 INTEGER*8 M, N, K, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE GEMM( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A, [LDA],
*       B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 INTEGER :: M, N, K, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C

 SUBROUTINE GEMM_64( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A,
*       [LDA], B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 INTEGER(8) :: M, N, K, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void sgemm(char transa, char transb, int m, int n, int k, float alpha, float *a, int lda, float *b, int ldb, float beta, float *c, int ldc);

void sgemm_64(char transa, char transb, long m, long n, long k, float alpha, float *a, long lda, float *b, long ldb, float beta, float *c, long ldc);

PURPOSE

sgemm performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C where op(X) is one of

```
op( X ) = X   or   op( X ) = X',
```

alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n', op(A) = A.

  TRANSA = 'T' or 't', op(A) = A'.

  TRANSA = 'C' or 'c', op(A) = A'.

  Unchanged on exit.
- **TRANSB (input)**
  On entry, TRANSB specifies the form of op(B) to be used in the matrix multiplication as follows:

  TRANSB = 'N' or 'n', op(B) = B.

  TRANSB = 'T' or 't', op(B) = B'.

  TRANSB = 'C' or 'c', op(B) = B'.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix op(A) and of the matrix C. M must be at least zero. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K must be at least zero. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is m otherwise. Before entry with TRANSA = 'N' or 'n', the leading m by k part of the array A must contain the matrix A, otherwise the leading k by m part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA > = max(1, m), otherwise LDA > = max(1, k). Unchanged on exit.
- **B (input)**
  n when TRANSB = 'N' or 'n', and is k otherwise. Before entry with TRANSB = 'N' or 'n', the leading k by n part of the array B must contain the matrix B, otherwise the leading n by k part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSB = 'N' or 'n' then LDB > = max(1, k), otherwise LDB > = max(1, n). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n matrix (alpha*op(A)*op(B) + beta*C).
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

sgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE SGEMV( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 INTEGER M, N, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)

 SUBROUTINE SGEMV_64( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 INTEGER*8 M, N, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE GEMV( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: M, N, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE GEMV_64( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: M, N, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgemv(char transa, int m, int n, float alpha, float *a, int lda, float *x, int incx, float beta, float *y, int incy);

void sgemv_64(char transa, long m, long n, float alpha, float *a, long lda, float *x, long incx, float beta, float *y, long incy);

PURPOSE

sgemv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*A'*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry with BETA non-zero, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

sgeqlf - compute a QL factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE SGEQLF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGEQLF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQLF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GEQLF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgeqlf(int m, int n, float *a, int lda, float *tau, int *info);

void sgeqlf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

sgeqlf computes a QL factorization of a real M-by-N matrix A: A = Q * L.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m >= n, the lower triangle of the subarray `A(m-n+1:m,1:n)` contains the N-by-N lower triangular matrix L; if m <= n, the elements on and below the (n-m)-th superdiagonal contain the M-by-N lower trapezoidal matrix L; the remaining elements, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(m-k+i+1:m)` = 0 and `v(m-k+i)` = 1; `v(1:m-k+i-1)` is stored on exit in A(1:m-k+i-1,n-k+i), and tau in TAU(i).

NAME

sgeqp3 - compute a QR factorization with column pivoting of a matrix A

SYNOPSIS

```
SUBROUTINE SGEQP3( M, N, A, LDA, JPVT, TAU, WORK, LWORK, INFO)
INTEGER M, N, LDA, LWORK, INFO
INTEGER JPVT(*)
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGEQP3_64( M, N, A, LDA, JPVT, TAU, WORK, LWORK, INFO)
INTEGER*8 M, N, LDA, LWORK, INFO
INTEGER*8 JPVT(*)
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQP3( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [INFO])
INTEGER :: M, N, LDA, LWORK, INFO
INTEGER, DIMENSION(:) :: JPVT
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GEQP3_64( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LWORK, INFO
INTEGER(8), DIMENSION(:) :: JPVT
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgeqp3(int m, int n, float *a, int lda, int *jpvt, float *tau, int *info);

void sgeqp3_64(long m, long n, float *a, long lda, long *jpvt, float *tau, long *info);

PURPOSE

sgeqp3 computes a QR factorization with column pivoting of a matrix A: A*P = Q*R using Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper trapezoidal matrix R; the elements below the diagonal, together with the array TAU, represent the orthogonal matrix Q as a product of `min(M,N)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **JPVT (input/output)**
  On entry, if JPVT(J).ne.0, the J-th column of A is permuted to the front of A*P (a leading column); if `JPVT(J)` =0, the J-th column of A is a free column. On exit, if `JPVT(J)` =K, then the J-th column of A*P was the the K-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 3*N+1. For optimal performance LWORK > = 2*N+(N+1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0: successful exit.

  < 0: if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real/complex scalar, and v is a real/complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

Based on contributions by

```
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
X. Sun, Computer Science Dept., Duke University, USA
```

NAME

sgeqpf - routine is deprecated and has been replaced by routine SGEQP3

SYNOPSIS

```
SUBROUTINE SGEQPF( M, N, A, LDA, JPIVOT, TAU, WORK, INFO)
INTEGER M, N, LDA, INFO
INTEGER JPIVOT(*)
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGEQPF_64( M, N, A, LDA, JPIVOT, TAU, WORK, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 JPIVOT(*)
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQPF( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: JPIVOT
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GEQPF_64( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: JPIVOT
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgeqpf(int m, int n, float *a, int lda, int *jpivot, float *tau, int *info);

void sgeqpf_64(long m, long n, float *a, long lda, long *jpivot, float *tau, long *info);

PURPOSE

sgeqpf routine is deprecated and has been replaced by routine SGEQP3.

SGEQPF computes a QR factorization with column pivoting of a real M-by-N matrix A: A*P = Q*R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper triangular matrix R; the elements below the diagonal, together with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is permuted to the front of A*P (a leading column); if `JPIVOT(i)` = 0, the i-th column of A is a free column. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n)
```

Each `H(i)` has the form

```
H  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i).

The matrix P is represented in jpvt as follows: If

```
jpvt(j)  = i
```

then the jth column of P is the ith canonical unit vector.

NAME

sgeqrf - compute a QR factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE SGEQRF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGEQRF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GEQRF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GEQRF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgeqrf(int m, int n, float *a, int lda, float *tau, int *info);

void sgeqrf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

sgeqrf computes a QR factorization of a real M-by-N matrix A: A = Q * R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and above the diagonal of the array contain the min(M,N)-by-N upper trapezoidal matrix R (R is upper triangular if m >= n); the elements below the diagonal, with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

NAME

sger - perform the rank 1 operation A := alpha*x*y' + A

SYNOPSIS

```
SUBROUTINE SGER( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
INTEGER M, N, INCX, INCY, LDA
REAL ALPHA
REAL X(*), Y(*), A(LDA,*)

SUBROUTINE SGER_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
INTEGER*8 M, N, INCX, INCY, LDA
REAL ALPHA
REAL X(*), Y(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GER( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
INTEGER :: M, N, INCX, INCY, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE GER_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
INTEGER(8) :: M, N, INCX, INCY, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sger(int m, int n, float alpha, float *x, int incx, float *y, int incy, float *a, int lda);

void sger_64(long m, long n, float alpha, float *x, long incx, float *y, long incy, float *a, long lda);

PURPOSE

sger performs the rank 1 operation A := alpha*x*y' + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

sgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SGERFS( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGERFS_64( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GERFS( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GERFS_64( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sgerfs(char transa, int n, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void sgerfs_64(char transa, long n, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

sgerfs improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original N-by-N matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGETRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sgerqf - compute an RQ factorization of a real M-by-N matrix A

SYNOPSIS

```
SUBROUTINE SGERQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SGERQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GERQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
INTEGER :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GERQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgerqf(int m, int n, float *a, int lda, float *tau, int *info);

void sgerqf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

sgerqf computes an RQ factorization of a real M-by-N matrix A: A = R * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m <= n, the upper triangle of the subarray `A(1:m,n-m+1:n)` contains the M-by-M upper triangular matrix R; if m >= n, the elements on and above the (m-n)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAU, represent the orthogonal matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and tau in TAU(i).

NAME

sgesdd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and right singular vectors

---

SYNOPSIS

```
 SUBROUTINE SGESDD( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER IWORK(*)
 REAL A(LDA,*), S(*), U(LDU,*), VT(LDVT,*), WORK(*)

 SUBROUTINE SGESDD_64( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER*8 M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL A(LDA,*), S(*), U(LDU,*), VT(LDVT,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESDD( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT, [LDVT],
*       [WORK], [LWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: S, WORK
 REAL, DIMENSION(:,:) :: A, U, VT

 SUBROUTINE GESDD_64( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT,
*       [LDVT], [WORK], [LWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: S, WORK
```

```
REAL, DIMENSION(:,:) :: A, U, VT
```

C INTERFACE

#include <sunperf.h>

void sgesdd(char jobz, int m, int n, float *a, int lda, float *s, float *u, int ldu, float *vt, int ldvt, int *info);

void sgesdd_64(char jobz, long m, long n, float *a, long lda, float *s, float *u, long ldu, float *vt, long ldvt, long *info);

PURPOSE

sgesdd computes the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and right singular vectors. If singular vectors are desired, it uses a divide-and-conquer algorithm.

The SVD is written

```
 = U * SIGMA * transpose(V)
```

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M orthogonal matrix, and V is an N-by-N orthogonal matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns VT = V**T, not V.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U and all N rows of V**T are
returned in the arrays U and VT;
 = 'S':  the first min(M,N) columns of U and the first
min(M,N) rows of V**T are returned in the arrays U
and VT;
 = 'O':  If M  > = N, the first N columns of U are overwritten
on the array A and all rows of V**T are returned in
the array VT;
otherwise, all columns of U are returned in the
array U and the first M rows of V**T are overwritten
in the array VT;
 = 'N':  no columns of U or rows of V**T are computed.
```

- **M (input)**
  The number of rows of the input matrix A. M > = 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBZ = 'O', A is overwritten with the first N columns of U (the left singular vectors, stored columnwise) if M >= N; A is overwritten with the first M rows of V**T (the right singular vectors, stored rowwise) otherwise. if JOBZ .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **S (output)**
  The singular values of A, sorted so that `S(i)` >= S(i+1).
- **U (output)**
  UCOL = M if JOBZ = 'A' or JOBZ = 'O' and M < N; UCOL = `min(M,N)` if JOBZ = 'S'. If JOBZ = 'A' or JOBZ = 'O' and M < N, U contains the M-by-M orthogonal matrix U; if JOBZ = 'S', U contains the first `min(M,N)` columns of U (the left singular vectors, stored columnwise); if JOBZ = 'O' and M >= N, or JOBZ = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBZ = 'S' or 'A' or JOBZ = 'O' and M < N, LDU >= M.
- **VT (output)**
  If JOBZ = 'A' or JOBZ = 'O' and M >= N, VT contains the N-by-N orthogonal matrix V**T; if JOBZ = 'S', VT contains the first `min(M,N)` rows of V**T (the right singular vectors, stored rowwise); if JOBZ = 'O' and M < N, or JOBZ = 'N', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBZ = 'A' or JOBZ = 'O' and M >= N, LDVT >= N; if JOBZ = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK;
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. If JOBZ = 'N', LWORK >= 3*min(M,N) + max(max(M,N),6*min(M,N)). If JOBZ = 'O', LWORK >= 3*min(M,N)*min(M,N) + max(max(M,N),5*min(M,N)* min(M,N)+4*min(M,N)). If JOBZ = 'S' or 'A' LWORK >= 3*min(M,N)*min(M,N) + max(max(M,N),4*min(M,N)* min(M,N)+4*min(M,N)). For good performance, LWORK should generally be larger. If LWORK < 0 but other input arguments are legal, `WORK(1)` returns optimal LWORK.
- **IWORK (workspace)**
  `dimension(8*MIN(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.

> 0:  SBDSDC did not converge, updating process failed.
```

FURTHER DETAILS

Based on contributions by

Ming Gu and Huan Ren, Computer Science Division, University of California at Berkeley, USA

NAME

sgesv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE SGESV( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SGESV_64( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GESV( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE GESV_64( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgesv(int n, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void sgesv_64(long n, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sgesv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as

```
A = P * L * U,
```

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N coefficient matrix A. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution could not be computed.
```

NAME

sgesvd - compute the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors

---

SYNOPSIS

```
 SUBROUTINE SGESVD( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBU, JOBVT
 INTEGER M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL A(LDA,*), SING(*), U(LDU,*), VT(LDVT,*), WORK(*)

 SUBROUTINE SGESVD_64( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, INFO)
 CHARACTER * 1 JOBU, JOBVT
 INTEGER*8 M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL A(LDA,*), SING(*), U(LDU,*), VT(LDVT,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVD( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 INTEGER :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL, DIMENSION(:) :: SING, WORK
 REAL, DIMENSION(:,:) :: A, U, VT

 SUBROUTINE GESVD_64( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL, DIMENSION(:) :: SING, WORK
 REAL, DIMENSION(:,:) :: A, U, VT
```

C INTERFACE

#include <sunperf.h>

void sgesvd(char jobu, char jobvt, int m, int n, float *a, int lda, float *sing, float *u, int ldu, float *vt, int ldvt, int *info);

void sgesvd_64(char jobu, char jobvt, long m, long n, float *a, long lda, float *sing, float *u, long ldu, float *vt, long ldvt, long *info);

PURPOSE

sgesvd computes the singular value decomposition (SVD) of a real M-by-N matrix A, optionally computing the left and/or right singular vectors. The SVD is written = U * SIGMA * `transpose(V)`

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M orthogonal matrix, and V is an N-by-N orthogonal matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns V**T, not V.

ARGUMENTS

- **JOBU (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U are returned in array U:

 = 'S':  the first min(m,n) columns of U (the left singular
vectors) are returned in the array U;
 = 'O':  the first min(m,n) columns of U (the left singular
vectors) are overwritten on the array A;
 = 'N':  no columns of U (no left singular vectors) are
computed.
```

- **JOBVT (input)**
  Specifies options for computing all or part of the matrix V**T:

```
 = 'A':  all N rows of V**T are returned in the array VT;

 = 'S':  the first min(m,n) rows of V**T (the right singular
vectors) are returned in the array VT;
 = 'O':  the first min(m,n) rows of V**T (the right singular
vectors) are overwritten on the array A;
 = 'N':  no rows of V**T (no right singular vectors) are
computed.
```

  JOBVT and JOBU cannot both be 'O'.

- **M (input)**
  The number of rows of the input matrix A. M > = 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBU = 'O', A is overwritten with the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBVT = 'O', A is overwritten with the first `min(m,n)` rows of V**T (the right singular vectors, stored rowwise); if JOBU .ne. 'O' and JOBVT .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **SING (output)**
  The singular values of A, sorted so that `SING(i)` >= SING(i+1).
- **U (output)**
  (LDU,M) if JOBU = 'A' or (LDU,min(M,N)) if JOBU = 'S'. If JOBU = 'A', U contains the M-by-M orthogonal matrix U; if JOBU = 'S', U contains the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBU = 'N' or 'O', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBU = 'S' or 'A', LDU >= M.
- **VT (output)**
  If JOBVT = 'A', VT contains the N-by-N orthogonal matrix V**T; if JOBVT = 'S', VT contains the first `min(m,n)` rows of V**T (the right singular vectors, stored rowwise); if JOBVT = 'N' or 'O', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBVT = 'A', LDVT >= N; if JOBVT = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK; if INFO > 0, `WORK(2:MIN(M,N))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in SING (not necessarily sorted). B satisfies A = U * B * VT, so it has the same singular values as A, and singular vectors related by U and VT.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1. LDWORK >= MAX(3*MIN(M,N)+MAX(M,N),5*MIN(M,N)). For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if SBDSQR did not converge, INFO specifies how many
superdiagonals of an intermediate bidiagonal form B
did not converge to zero. See the description of WORK
above for details.
```

NAME

sgesvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SGESVX( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR, WORK,
*      WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGESVX_64( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF,
*      IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), ROWSC(*), COLSC(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVX( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE GESVX_64( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND,
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sgesvx(char fact, char transa, int n, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, char equed, float *rowsc, float *colsc, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sgesvx_64(char fact, char transa, long n, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, char equed, float *rowsc, float *colsc, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sgesvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = P * L * U,

where P is a permutation matrix, L is a unit lower triangular
matrix, and U is upper triangular.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

```
 = 'E':  The matrix A will be equilibrated if necessary, then
copied to AF and factored.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'COLSC':  A**H * X  = B  (Transpose)
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the factors L and U from the factorization A = P*L*U as computed by SGETRF. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = P*L*U as computed by SGETRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the equilibrated matrix A.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(4*N)` On exit, `WORK(1)` contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If `WORK(1)` is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then `WORK(1)` contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization has
been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

sgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE SGETF2( M, N, A, LDA, IPIV, INFO)
INTEGER M, N, LDA, INFO
INTEGER IPIV(*)
REAL A(LDA,*)

SUBROUTINE SGETF2_64( M, N, A, LDA, IPIV, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIV(*)
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GETF2( [M], [N], A, [LDA], IPIV, [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: A

SUBROUTINE GETF2_64( [M], [N], A, [LDA], IPIV, [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgetf2(int m, int n, float *a, int lda, int *ipiv, int *info);

void sgetf2_64(long m, long n, float *a, long lda, long *ipiv, long *info);

PURPOSE

sgetf2 computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 2 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the m by n matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

sgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE SGETRF( M, N, A, LDA, IPIVOT, INFO)
INTEGER M, N, LDA, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*)

SUBROUTINE SGETRF_64( M, N, A, LDA, IPIVOT, INFO)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE GETRF( [M], [N], A, [LDA], IPIVOT, [INFO])
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A

SUBROUTINE GETRF_64( [M], [N], A, [LDA], IPIVOT, [INFO])
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgetrf(int m, int n, float *a, int lda, int *ipivot, int *info);

void sgetrf_64(long m, long n, float *a, long lda, long *ipivot, long *info);

PURPOSE

sgetrf computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 3 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

sgetri - compute the inverse of a matrix using the LU factorization computed by SGETRF

SYNOPSIS

```
SUBROUTINE SGETRI( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), WORK(*)

SUBROUTINE SGETRI_64( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE GETRI( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE GETRI_64( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sgetri(int n, float *a, int lda, int *ipivot, int *info);

void sgetri_64(long n, float *a, long lda, long *ipivot, long *info);

PURPOSE

sgetri computes the inverse of a matrix using the LU factorization computed by SGETRF.

This method inverts U and then computes `inv(A)` by solving the system inv(A)*L = `inv(U)` for inv(A).

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the factors L and U from the factorization A = P*L*U as computed by SGETRF. On exit, if INFO = 0, the inverse of the original matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **WORK (workspace)**
  On exit, if INFO =0, then `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimal performance LDWORK > = N*NB, where NB is the optimal blocksize returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero; the matrix is
singular and its inverse could not be computed.
```

NAME

sgetrs - solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF

SYNOPSIS

```
SUBROUTINE SGETRS( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SGETRS_64( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GETRS( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: TRANSA
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE GETRS_64( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
CHARACTER(LEN=1) :: TRANSA
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgetrs(char transa, int n, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void sgetrs_64(char transa, long n, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sgetrs solves a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by SGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from SGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sggbak - form the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL

SYNOPSIS

```
 SUBROUTINE SGGBAK( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER N, ILO, IHI, M, LDV, INFO
 REAL LSCALE(*), RSCALE(*), V(LDV,*)

 SUBROUTINE SGGBAK_64( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 REAL LSCALE(*), RSCALE(*), V(LDV,*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAK( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M], V,
*       [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE
 REAL, DIMENSION(:,:) :: V

 SUBROUTINE GGBAK_64( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M],
*       V, [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE
 REAL, DIMENSION(:,:) :: V
```

C INTERFACE

#include <sunperf.h>

void sggbak(char job, char side, int n, int ilo, int ihi, float *lscale, float *rscale, int m, float *v, int ldv, int *info);

void sggbak_64(char job, char side, long n, long ilo, long ihi, float *lscale, float *rscale, long m, float *v, long ldv, long *info);

PURPOSE

sggbak forms the right or left eigenvectors of a real generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by SGGBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required:

```
 = 'N':  do nothing, return immediately;

 = 'P':  do backward transformation for permutation only;

 = 'S':  do backward transformation for scaling only;

 = 'B':  do backward transformations for both permutation and
scaling.
JOB must be the same as the argument JOB supplied to SGGBAL.
```

- **SIDE (input)**

```
 = 'R':  V contains right eigenvectors;

 = 'L':  V contains left eigenvectors.
```

- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by SGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description for ILO.
- **LSCALE (input)**
  Details of the permutations and/or scaling factors applied to the left side of A and B, as returned by SGGBAL.
- **RSCALE (input)**
  Details of the permutations and/or scaling factors applied to the right side of A and B, as returned by SGGBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**

On entry, the matrix of right or left eigenvectors to be transformed, as returned by STGEVC. On exit, V is overwritten by the transformed eigenvectors.

- **LDV (input)**
  The leading dimension of the matrix V. LDV >= max(1,N).
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. Ward, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

sggbal - balance a pair of general real matrices (A,B)

SYNOPSIS

```
 SUBROUTINE SGGBAL( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE, RSCALE,
*      WORK, INFO)
 CHARACTER * 1 JOB
 INTEGER N, LDA, LDB, ILO, IHI, INFO
 REAL A(LDA,*), B(LDB,*), LSCALE(*), RSCALE(*), WORK(*)

 SUBROUTINE SGGBAL_64( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE,
*      RSCALE, WORK, INFO)
 CHARACTER * 1 JOB
 INTEGER*8 N, LDA, LDB, ILO, IHI, INFO
 REAL A(LDA,*), B(LDB,*), LSCALE(*), RSCALE(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAL( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 INTEGER :: N, LDA, LDB, ILO, IHI, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GGBAL_64( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 INTEGER(8) :: N, LDA, LDB, ILO, IHI, INFO
 REAL, DIMENSION(:) :: LSCALE, RSCALE, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sggbal(char job, int n, float *a, int lda, float *b, int ldb, int *ilo, int *ihi, float *lscale, float *rscale, int *info);

void sggbal_64(char job, long n, float *a, long lda, float *b, long ldb, long *ilo, long *ihi, float *lscale, float *rscale, long *info);

PURPOSE

sggbal balances a pair of general real matrices (A,B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to ILO$-$1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem A*x = lambda*B*x.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A and B:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, LSCALE(I)  = 1.0
and RSCALE(I)  = 1.0 for i  = 1,...,N.
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the input matrix B. On exit, B is overwritten by the balanced matrix. If JOB = 'N', B is not referenced.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description for ILO.
- **LSCALE (input)**

Details of the permutations and scaling factors applied to the left side of A and B. If `P(j)` is the index of the row interchanged with row j, and `D(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.

- **RSCALE (input)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `P(j)` is the index of the column interchanged with column j, and `D(j)` is the scaling factor applied to column j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. WARD, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

sgges - compute for a pair of N-by-N real nonsymmetric matrices (A,B),

---

SYNOPSIS

```
 SUBROUTINE SGGES( JOBVSL, JOBVSR, SORT, SELCTG, N, A, LDA, B, LDB,
*      SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK,
*      BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL SELCTG
 LOGICAL BWORK(*)
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)

 SUBROUTINE SGGES_64( JOBVSL, JOBVSR, SORT, SELCTG, N, A, LDA, B,
*      LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR, WORK,
*      LWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL*8 SELCTG
 LOGICAL*8 BWORK(*)
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGES( JOBVSL, JOBVSR, SORT, SELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       [WORK], [LWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL :: SELCTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR

 SUBROUTINE GGES_64( JOBVSL, JOBVSR, SORT, SELCTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       [WORK], [LWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL(8) :: SELCTG
 LOGICAL(8), DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void sgges(char jobvsl, char jobvsr, char sort, logical(*selctg)(float,float,float), int n, float *a, int lda, float *b, int ldb, int *sdim, float *alphar, float *alphai, float *beta, float *vsl, int ldvsl, float *vsr, int ldvsr, int *info);

void sgges_64(char jobvsl, char jobvsr, char sort, logical(*selctg)(float,float,float), long n, float *a, long lda, float *b, long ldb, long *sdim, float *alphar, float *alphai, float *beta, float *vsl, long ldvsl, float *vsr, long ldvsr, long *info);

PURPOSE

sgges computes for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized real Schur form (S,T), optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization

```
(A,B) = ( (VSL)*S*(VSR)**T, (VSL)*T*(VSR)**T )
```

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T.The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver SGGEV instead, which is faster.)

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or both being zero.

A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be ``standardized" by making the corresponding elements of T have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues.

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

  ```
  = 'S':  Eigenvalues are ordered (see SELCTG);
  ```

- **SELCTG (input)**
  SELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', SELCTG is not referenced. If SORT = 'S', SELCTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue (ALPHAR(j)+ALPHAI(j))/BETA(j) is selected if `SELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` is true; i.e. if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

  Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy SELCTG(ALPHAR(j),ALPHAI(j), `BETA(j))` = .TRUE. after ordering. INFO is to be set to N+2 in this case.
- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which SELCTG is true. (Complex conjugate pairs for which SELCTG is true for either eigenvalue count as 2.)
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i, and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > =1, and if JOBVSL = 'V', LDVSL > = N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR > = 1, and if JOBVSR = 'V', LDVSR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 8*N+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

 = 1,...,N:
The QZ iteration failed. (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N: =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy SELCTG =.TRUE. This could also
be caused due to scaling.
 =N+3: reordering failed in STGSEN.

NAME

sggesx - compute for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and,

---

SYNOPSIS

```
 SUBROUTINE SGGESX( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, N, A, LDA,
*      B, LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR,
*      RCONDE, RCONDV, WORK, LWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELCTG
 LOGICAL BWORK(*)
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), RCONDE(*), RCONDV(*), WORK(*)

 SUBROUTINE SGGESX_64( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, N, A,
*      LDA, B, LDB, SDIM, ALPHAR, ALPHAI, BETA, VSL, LDVSL, VSR, LDVSR,
*      RCONDE, RCONDV, WORK, LWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELCTG
 LOGICAL*8 BWORK(*)
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), RCONDE(*), RCONDV(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGESX( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, [N], A, [LDA],
*       B, [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       RCONDE, RCONDV, [WORK], [LWORK], [IWORK], [LIWORK], [BWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL :: SELCTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, RCONDE, RCONDV, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR

 SUBROUTINE GGESX_64( JOBVSL, JOBVSR, SORT, SELCTG, SENSE, [N], A,
*       [LDA], B, [LDB], SDIM, ALPHAR, ALPHAI, BETA, VSL, [LDVSL], VSR,
*       [LDVSR], RCONDE, RCONDV, [WORK], [LWORK], [IWORK], [LIWORK],
*       [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
 INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8) :: SELCTG
 LOGICAL(8), DIMENSION(:) :: BWORK
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, RCONDE, RCONDV, WORK
 REAL, DIMENSION(:,:) :: A, B, VSL, VSR
```

C INTERFACE

#include <sunperf.h>

void sggesx(char jobvsl, char jobvsr, char sort, logical(*selctg)(float,float,float), char sense, int n, float *a, int lda, float *b, int ldb, int *sdim, float *alphar, float *alphai, float *beta, float *vsl, int ldvsl, float *vsr, int ldvsr, float *rconde, float *rcondv, int *info);

void sggesx_64(char jobvsl, char jobvsr, char sort, logical(*selctg)(float,float,float), char sense, long n, float *a, long lda, float *b, long ldb, long *sdim, float *alphar, float *alphai, float *beta, float *vsl, long ldvsl, float *vsr, long ldvsr, float *rconde, float *rcondv, long *info);

PURPOSE

sggesx computes for a pair of N-by-N real nonsymmetric matrices (A,B), the generalized eigenvalues, the real Schur form (S,T), and, optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization

A,B) = ((VSL) S (VSR)**T, (VSL) T (VSR)**T)

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (RCONDV). The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or for both being zero.

A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be ``standardized'' by making the corresponding elements of T have the form:

```
[  a  0  ]

[  0  b  ]
```

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues.

ARGUMENTS

- **JOBVSL (input)**

```
= 'N':  do not compute the left Schur vectors;

= 'V':  compute the left Schur vectors.
```

- **JOBVSR (input)**

```
= 'N':  do not compute the right Schur vectors;

= 'V':  compute the right Schur vectors.
```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S':  Eigenvalues are ordered (see SELCTG).
```

- **SELCTG (input)**
  SELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', SELCTG is not referenced. If SORT = 'S', SELCTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue (ALPHAR(j)+ALPHAI(j))/BETA(j) is selected if `SELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` is true; i.e. if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy `SELCTG(ALPHAR(j),ALPHAI(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+3.
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N' : None are computed;

```
= 'E' : Computed for average of selected eigenvalues only;

= 'V' : Computed for selected deflating subspaces only;

= 'B' : Computed for both.
If SENSE  = 'E', 'V', or 'B', SORT must equal 'S'.
```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.

- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which SELCTG is true. (Complex conjugate pairs for which SELCTG is true for either eigenvalue count as 2.)
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real Schur form of (A,B) were further reduced to triangular form using 2-by-2 complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >=1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', `RCONDE(1)` and `RCONDE(2)` contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', `RCONDV(1)` and `RCONDV(2)` contain the reciprocal condition numbers for the selected deflating subspaces. Not referenced if SENSE = 'N' or 'E'.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 8*(N+1)+16. If SENSE = 'E', 'V', or 'B', LWORK >= MAX(8*(N+1)+16, 2*SDIM*(N-SDIM)).
- **IWORK (workspace)**
  Not referenced if SENSE = 'N'.
- **LIWORK (input)**
  The dimension of the array WORK. LIWORK >= N+6.
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHAR(j), ALPHAI(j), and BETA(j) should
be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy SELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering failed in STGSEN.
```

Further details = = = = = = = = = = = = = = =

An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

EPS * norm((A, B)) / RCONDE(1).

An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is

EPS * norm((A, B)) / RCONDV(2).

See LAPACK User's Guide, section 4.11 for more information.

NAME

sggev - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

SYNOPSIS

```
 SUBROUTINE SGGEV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR, ALPHAI,
*      BETA, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE SGGEV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHAR,
*      ALPHAI, BETA, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGEV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE GGEV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHAR,
*       ALPHAI, BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void sggev(char jobvl, char jobvr, int n, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *vl, int ldvl, float *vr, int ldvr, int *info);

void sggev_64(char jobvl, char jobvr, long n, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *vl, long ldvl, float *vr, long ldvr, long *info);

PURPOSE

sggev computes for a pair of N-by-N real nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j).
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B .
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)`

may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).

- **ALPHAI (output)**
  See the description for ALPHAR.
- **BETA (output)**
  See the description for ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)+i*VL(:,j+1)` and `u(j+1)` = VL(:,j)-i*VL(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part)+abs(imag. part) =1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)+i*VR(:,j+1)` and `v(j+1)` = VR(:,j)-i*VR(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part)+abs(imag. part) =1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,8*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: error return from STGEVC.
```

NAME

sggevx - compute for a pair of N-by-N real nonsymmetric matrices (A,B)

SYNOPSIS

```
  SUBROUTINE SGGEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B, LDB,
 *      ALPHAR, ALPHAI, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE,
 *      RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, BWORK,
 *      INFO)
  CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
  INTEGER N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER IWORK(*)
  LOGICAL BWORK(*)
  REAL ABNRM, BBNRM
  REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), LSCALE(*), RSCALE(*), RCONDE(*),
RCONDV(*), WORK(*)

  SUBROUTINE SGGEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B,
 *      LDB, ALPHAR, ALPHAI, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE,
 *      RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, BWORK,
 *      INFO)
  CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
  INTEGER*8 N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER*8 IWORK(*)
  LOGICAL*8 BWORK(*)
  REAL ABNRM, BBNRM
  REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), VL(LDVL,*), VR(LDVR,*), LSCALE(*), RSCALE(*), RCONDE(*),
RCONDV(*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE GGEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHAR, ALPHAI, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI,
 *       LSCALE, RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  INTEGER :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL, DIMENSION(:) :: BWORK
  REAL :: ABNRM, BBNRM
  REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, LSCALE, RSCALE, RCONDE, RCONDV, WORK
  REAL, DIMENSION(:,:) :: A, B, VL, VR

  SUBROUTINE GGEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHAR, ALPHAI, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI,
 *       LSCALE, RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL :: ABNRM, BBNRM
  REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, LSCALE, RSCALE, RCONDE, RCONDV, WORK
  REAL, DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void sggevx(char balanc, char jobvl, char jobvr, char sense, int n, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *vl, int ldvl, float *vr, int ldvr, int *ilo, int *ihi, float *lscale, float *rscale, float *abnrm, float *bbnrm, float *rconde, float *rcondv, int *info);

void sggevx_64(char balanc, char jobvl, char jobvr, char sense, long n, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *vl, long ldvl, float *vr, long ldvr, long *ilo, long *ihi, float *lscale, float *rscale, float *abnrm, float *bbnrm, float *rconde, float *rcondv, long *info);

PURPOSE

sggevx computes for a pair of N-by-N real nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, LSCALE, RSCALE, ABNRM, and BBNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right eigenvectors (RCONDV).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j) .
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B.
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **BALANC (input)**
  Specifies the balance option to be performed. = 'N': do not diagonally scale or permute;

  ```
  = 'P':  permute only;

  = 'S':  scale only;

  = 'B':  both permute and scale.
  Computed reciprocal condition numbers will be for the
  matrices after permuting and/or balancing. Permuting does
  not change condition numbers (in exact arithmetic), but
  balancing does.
  ```
- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```
- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': none are computed;

  ```
  = 'E': computed for eigenvalues only;

  = 'V': computed for eigenvectors only;

  = 'B': computed for eigenvalues and eigenvectors.
  ```
- **N (input)**
  The order of the matrices A, B, VL, and VR. N >= 0.
- **A (input/output)**

On entry, the matrix A in the pair (A,B). On exit, A has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then A contains the first part of the real Schur form of the ``balanced'' versions of the input A and B.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then B contains the second part of the real Schur form of the ``balanced'' versions of the input A and B.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

  Note: the quotients `ALPHAR(j)/BETA(j)` and `ALPHAI(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio ALPHA/BETA. However, ALPHAR and ALPHAI will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `u(j)` = VL(:,j), the j-th column of VL. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `u(j)` = `VL(:,j)+i*VL(:,j+1)` and `u(j+1)` = VL(:,j)-i*VL(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL > = 1, and if JOBVL = 'V', LDVL > = N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If the j-th eigenvalue is real, then `v(j)` = VR(:,j), the j-th column of VR. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then `v(j)` = `VR(:,j)+i*VR(:,j+1)` and `v(j+1)` = VR(:,j)-i*VR(:,j+1). Each eigenvector will be scaled so the largest component have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR > = 1, and if JOBVR = 'V', LDVR > = N.
- **ILO (output)**
  ILO and IHI are integer values such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  See the description of ILO.
- **LSCALE (output)**
  Details of the permutations and scaling factors applied to the left side of A and B. If `PL(j)` is the index of the row interchanged with row j, and `DL(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `PL(j)` for j = 1,...,ILO-1 = `DL(j)` for j = ILO,...,IHI = `PL(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (output)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `PR(j)` is the index of the column interchanged with column j, and `DR(j)` is the scaling factor applied to column j, then `RSCALE(j)` = `PR(j)` for j = 1,...,ILO-1 = `DR(j)` for j = ILO,...,IHI = `PR(j)` for j = IHI+1,...,N The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix A.
- **BBNRM (output)**
  The one-norm of the balanced matrix B.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of RCONDE are set to the same value. Thus RCONDE(j), RCONDV(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If SENSE = 'V', RCONDE is not referenced.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of RCONDV are set to the same value. If the eigenvalues cannot be reordered to compute RCONDV(j), `RCONDV(j)` is set to 0; this can only occur when the true value would be very small anyway. If SENSE = 'E', RCONDV is not referenced.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,6*N). If SENSE = 'E', LWORK > = 12*N. If SENSE = 'V' or 'B', LWORK > = 2*N*N+12*N+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(N+6)` If SENSE = 'E', IWORK is not referenced.
- **BWORK (workspace)**
  `dimension(N)` If SENSE = 'N', BWORK is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHAR(j), ALPHAI(j), and BETA(j)
should be correct for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in SHGEQZ.

 =N+2: error return from STGEVC.
```

FURTHER DETAILS

Balancing a matrix pair (A,B) includes, first, permuting rows and columns to isolate eigenvalues, second, applying diagonal similarity transformation to the rows and columns to make the rows and columns as close in norm as possible. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.11.1.2 of LAPACK Users' Guide.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(ABNRM, BBNRM) / `RCONDE(I)`

An approximate error bound for the angle between the i-th computed eigenvector `VL(i)` or `VR(i)` is given by

PS * norm(ABNRM, BBNRM) / DIF(i).

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see section 4.11 of LAPACK User's Guide.

NAME

sggglm - solve a general Gauss-Markov linear model (GLM) problem

---

SYNOPSIS

```
 SUBROUTINE SGGGLM( N, M, P, A, LDA, B, LDB, D, X, Y, WORK, LDWORK,
*      INFO)
 INTEGER N, M, P, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)

 SUBROUTINE SGGGLM_64( N, M, P, A, LDA, B, LDB, D, X, Y, WORK,
*      LDWORK, INFO)
 INTEGER*8 N, M, P, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGGLM( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y, [WORK],
*       [LDWORK], [INFO])
 INTEGER :: N, M, P, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: D, X, Y, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GGGLM_64( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y,
*       [WORK], [LDWORK], [INFO])
 INTEGER(8) :: N, M, P, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: D, X, Y, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sggglm(int n, int m, int p, float *a, int lda, float *b, int ldb, float *d, float *x, float *y, int *info);

void sggglm_64(long n, long m, long p, float *a, long lda, float *b, long ldb, float *d, float *x, float *y, long *info);

PURPOSE

sggglm solves a general Gauss-Markov linear model (GLM) problem:

```
        minimize || y ||_2   subject to   d = A*x + B*y

            x
```

where A is an N-by-M matrix, B is an N-by-P matrix, and d is a given N-vector. It is assumed that M <= N <= M+P, and

```
           rank(A) = M    and    rank( A B ) = N.
```

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of A and B.

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

```
             minimize || inv(B)*(d-A*x) ||_2

                 x
```

where `inv(B)` denotes the inverse of B.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. 0 < = M < = N.
- **P (input)**
  The number of columns of the matrix B. P > = N-M.
- **A (input/output)**
  On entry, the N-by-M matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-P matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **D (input/output)**
  On entry, D is the left hand side of the GLM equation. On exit, D is destroyed.
- **X (output)**
  On exit, X and Y are the solutions of the GLM problem.
- **Y (output)**
  See the description of X.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N+M+P). For optimum performance, LDWORK >= M+min(N,P)+max(N,P)*NB, where NB is an upper bound for the optimal blocksizes for SGEQRF, SGERQF, SORMQR and SORMRQ.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

sgghrd - reduce a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular

---

SYNOPSIS

```
 SUBROUTINE SGGHRD( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)

 SUBROUTINE SGGHRD_64( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
```

F95 INTERFACE

```
 SUBROUTINE GGHRD( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL, DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE GGHRD_64( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
 REAL, DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void sgghrd(char compq, char compz, int n, int ilo, int ihi, float *a, int lda, float *b, int ldb, float *q, int ldq, float *z, int ldz, int *info);

void sgghrd_64(char compq, char compz, long n, long ilo, long ihi, float *a, long lda, float *b, long ldb, float *q, long ldq, float *z, long ldz, long *info);

PURPOSE

sgghrd reduces a pair of real matrices (A,B) to generalized upper Hessenberg form using orthogonal transformations, where A is a general matrix and B is upper triangular: Q' * A * Z = H and Q' * B * Z = T, where H is upper Hessenberg, T is upper triangular, and Q and Z are orthogonal, and ' means transpose.

The orthogonal matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q1 and Z1, so that

1 * A * Z1' = (Q1*Q) * H * (Z1*Z)' 1 * B * Z1' = (Q1*Q) * T * (Z1*Z)'

ARGUMENTS

- **COMPQ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
orthogonal matrix Q is returned;
 = 'V': Q must contain an orthogonal matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **COMPZ (input)**

```
 = 'N': do not compute Z;

 = 'I': Z is initialized to the unit matrix, and the
orthogonal matrix Z is returned;
 = 'V': Z must contain an orthogonal matrix Z1 on entry,
and the product Z1*Z is returned.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGGBAL; otherwise they should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**

On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the rest is set to zero.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. On exit, the upper triangular matrix T = Q' B Z. The elements below the diagonal are set to zero.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **Q (input/output)**
  If COMPQ ='N': Q is not referenced.

  If COMPQ ='I': on entry, Q need not be set, and on exit it contains the orthogonal matrix Q, where Q' is the product of the Givens transformations which are applied to A and B on the left. If COMPQ ='V': on entry, Q must contain an orthogonal matrix Q1, and on exit this is overwritten by Q1*Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = N if COMPQ ='V' or 'I'; LDQ > = 1 otherwise.
- **Z (input/output)**
  If COMPZ ='N': Z is not referenced.

  If COMPZ ='I': on entry, Z need not be set, and on exit it contains the orthogonal matrix Z, which is the product of the Givens transformations which are applied to A and B on the right. If COMPZ ='V': on entry, Z must contain an orthogonal matrix Z1, and on exit this is overwritten by Z1*Z.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = N if COMPZ ='V' or 'I'; LDZ > = 1 otherwise.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

This routine reduces A to Hessenberg and B to triangular form by an unblocked reduction, as described in _Matrix_Computations_, by Golub and Van Loan (Johns Hopkins Press.)

NAME

sgglse - solve the linear equality-constrained least squares (LSE) problem

---

SYNOPSIS

```
 SUBROUTINE SGGLSE( M, N, P, A, LDA, B, LDB, C, D, X, WORK, LDWORK,
*      INFO)
 INTEGER M, N, P, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)

 SUBROUTINE SGGLSE_64( M, N, P, A, LDA, B, LDB, C, D, X, WORK,
*      LDWORK, INFO)
 INTEGER*8 M, N, P, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGLSE( [M], [N], [P], A, [LDA], B, [LDB], C, D, X, [WORK],
*       [LDWORK], [INFO])
 INTEGER :: M, N, P, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: C, D, X, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GGLSE_64( [M], [N], [P], A, [LDA], B, [LDB], C, D, X,
*       [WORK], [LDWORK], [INFO])
 INTEGER(8) :: M, N, P, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: C, D, X, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sgglse(int m, int n, int p, float *a, int lda, float *b, int ldb, float *c, float *d, float *x, int *info);

void sgglse_64(long m, long n, long p, float *a, long lda, float *b, long ldb, float *c, float *d, float *x, long *info);

PURPOSE

sgglse solves the linear equality-constrained least squares (LSE) problem:

```
minimize || c - A*x ||_2   subject to   B*x = d
```

where A is an M-by-N matrix, B is a P-by-N matrix, c is a given M-vector, and d is a given P-vector. It is assumed that

P <= N <= M+P, and

```
rank(B) = P and  rank( ( A ) ) = N.
                       ( ( B ) )
```

These conditions ensure that the LSE problem has a unique solution, which is obtained using a GRQ factorization of the matrices B and A.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **P (input)**
  The number of rows of the matrix B. 0 < = P < = N < = M+P.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **C (input/output)**
  On entry, C contains the right hand side vector for the least squares part of the LSE problem. On exit, the residual sum of squares for the solution is given by the sum of squares of elements N-P+1 to M of vector C.
- **D (input/output)**
  On entry, D contains the right hand side vector for the constrained equation. On exit, D is destroyed.
- **X (output)**
  On exit, X is the solution of the LSE problem.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,M+N+P). For optimum performance LDWORK > = P+min(M,N)+max(M,N)*NB, where NB is an upper bound for the optimal blocksizes for SGEQRF, SGERQF, SORMQR and SORMRQ.

If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

sggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

SYNOPSIS

```
 SUBROUTINE SGGQRF( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*       INFO)
 INTEGER N, M, P, LDA, LDB, LWORK, INFO
 REAL A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)

 SUBROUTINE SGGQRF_64( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK,
*       LWORK, INFO)
 INTEGER*8 N, M, P, LDA, LDB, LWORK, INFO
 REAL A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGQRF( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*        [WORK], [LWORK], [INFO])
 INTEGER :: N, M, P, LDA, LDB, LWORK, INFO
 REAL, DIMENSION(:) :: TAUA, TAUB, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GGQRF_64( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*        [WORK], [LWORK], [INFO])
 INTEGER(8) :: N, M, P, LDA, LDB, LWORK, INFO
 REAL, DIMENSION(:) :: TAUA, TAUB, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sggqrf(int n, int m, int p, float *a, int lda, float *taua, float *b, int ldb, float *taub, int *info);

void sggqrf_64(long n, long m, long p, float *a, long lda, float *taua, float *b, long ldb, float *taub, long *info);

PURPOSE

sggqrf computes a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B:

```
A = Q*R,        B = Q*T*Z,
```

where Q is an N-by-N orthogonal matrix, Z is a P-by-P orthogonal matrix, and R and T assume one of the forms:

if N >= M, R = (R11) M , or if N < M, R = (R11 R12) N, (0) N-M N M-N M

where R11 is upper triangular, and

if N <= P, T = (0 T12) N, or if N > P, T = (T11) N-P, P-N N (T21) P P

where T12 or T21 is upper triangular.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of inv(B)*A:

```
inv(B)*A = Z'*(inv(T)*R)
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the transpose of the matrix Z.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. M > = 0.
- **P (input)**
  The number of columns of the matrix B. P > = 0.
- **A (input)**
  On entry, the N-by-M matrix A. On exit, the elements on and above the diagonal of the array contain the min(N,M)-by-M upper trapezoidal matrix R (R is upper triangular if N > = M); the elements below the diagonal, with the array TAUA, represent the orthogonal matrix Q as a product of `min(N,M)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q (see Further Details).
- **B (input)**
  On entry, the N-by-P matrix B. On exit, if N < = P, the upper triangle of the subarray `B(1:N,P-N+1:P)` contains the N-by-N upper triangular matrix T; if N > P, the elements on and above the (N-P)-th subdiagonal contain the N-by-P upper trapezoidal matrix T; the remaining elements, with the array TAUB, represent the orthogonal matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the QR factorization of an N-by-M matrix, NB2 is the optimal blocksize for the RQ factorization of an N-by-P matrix, and NB3 is the optimal blocksize for a call of SORMQR.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(n,m).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i+1:n,i), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine SORGQR.

To use Q to update another matrix, use LAPACK subroutine SORMQR.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(n,p).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a real scalar, and v is a real vector with

`v(p-k+i+1:p)` = 0 and `v(p-k+i)` = 1; `v(1:p-k+i-1)` is stored on exit in B(n-k+i,1:p-k+i-1), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine SORGRQ.

To use Z to update another matrix, use LAPACK subroutine SORMRQ.

NAME

sggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

SYNOPSIS

```
 SUBROUTINE SGGRQF( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 INTEGER M, P, N, LDA, LDB, LWORK, INFO
 REAL A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)

 SUBROUTINE SGGRQF_64( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 INTEGER*8 M, P, N, LDA, LDB, LWORK, INFO
 REAL A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGRQF( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER :: M, P, N, LDA, LDB, LWORK, INFO
 REAL, DIMENSION(:) :: TAUA, TAUB, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE GGRQF_64( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 INTEGER(8) :: M, P, N, LDA, LDB, LWORK, INFO
 REAL, DIMENSION(:) :: TAUA, TAUB, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sggrqf(int m, int p, int n, float *a, int lda, float *taua, float *b, int ldb, float *taub, int *info);

void sggrqf_64(long m, long p, long n, float *a, long lda, float *taua, float *b, long ldb, float *taub, long *info);

PURPOSE

sggrqf computes a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B:

```
A = R*Q,        B = Z*T*Q,
```

where Q is an N-by-N orthogonal matrix, Z is a P-by-P orthogonal matrix, and R and T assume one of the forms:

if M <= N, R = (0 R12) M, or if M > N, R = (R11) M-N, N-M M (R21) N N

where R12 or R21 is upper triangular, and

if P >= N, T = (T11) N , or if P < N, T = (T11 T12) P, (0) P-N P N-P N

where T11 is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of A*inv(B):

```
A*inv(B) = (R*inv(T))*Z'
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the transpose of the matrix Z.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **P (input)**
  The number of rows of the matrix B. P > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M < = N, the upper triangle of the subarray `A(1:M,N-M+1:N)` contains the M-by-M upper triangular matrix R; if M > N, the elements on and above the (M-N)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAUA, represent the orthogonal matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Q (see Further Details).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, the elements on and above the diagonal of the array contain the min(P,N)-by-N upper trapezoidal matrix T (T is upper triangular if P > = N); the elements below the diagonal, with the array TAUB, represent the orthogonal matrix Z as a product of elementary reflectors (see Further Details).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).

- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the orthogonal matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the RQ factorization of an M-by-N matrix, NB2 is the optimal blocksize for the QR factorization of a P-by-N matrix, and NB3 is the optimal blocksize for a call of SORMRQ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INF0 = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a real scalar, and v is a real vector with

`v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine SORGRQ.

To use Q to update another matrix, use LAPACK subroutine SORMRQ.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(p,n).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a real scalar, and v is a real vector with

`v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:p)` is stored on exit in B(i+1:p,i), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine SORGQR.

To use Z to update another matrix, use LAPACK subroutine SORMQR.

NAME

sggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B

SYNOPSIS

```
 SUBROUTINE SGGSVD( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B, LDB,
*      ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER IWORK3(*)
 REAL A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)

 SUBROUTINE SGGSVD_64( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B,
*      LDB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK3(*)
 REAL A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVD( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK3
 REAL, DIMENSION(:) :: ALPHA, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE GGSVD_64( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER(8) :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3
 REAL, DIMENSION(:) :: ALPHA, BETA, WORK
```

```
REAL, DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void sggsvd(char jobu, char jobv, char jobq, int m, int n, int p, int *k, int *l, float *a, int lda, float *b, int ldb, float *alpha, float *beta, float *u, int ldu, float *v, int ldv, float *q, int ldq, int *iwork3, int *info);

void sggsvd_64(char jobu, char jobv, char jobq, long m, long n, long p, long *k, long *l, float *a, long lda, float *b, long ldb, float *alpha, float *beta, float *u, long ldu, float *v, long ldv, float *q, long ldq, long *iwork3, long *info);

PURPOSE

sggsvd computes the generalized singular value decomposition (GSVD) of an M-by-N real matrix A and P-by-N real matrix B:

```
    U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R )
```

where U, V and Q are orthogonal matrices, and Z' is the transpose of Z. Let K+L = the effective numerical rank of the matrix (A',B')', then R is a K+L-by-K+L nonsingular upper triangular matrix, D1 and D2 are M-by-(K+L) and P-by-(K+L) ``diagonal'' matrices and of the following structures, respectively:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
             M-K-L ( 0  0 )

                  K  L
       D2 =   L ( 0  S )
             P-L ( 0  0 )

                 N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 )
            L (  0    0   R22 )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),

  S = diag( BETA(K+1),  ... , BETA(K+L) ),
```

```
  C**2 + S**2 = I.

  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                   K M-K K+L-M

        D1 =   K ( I  0    0   )

             M-K ( 0  C    0   )

                     K M-K K+L-M

        D2 =   M-K ( 0  S    0  )

             K+L-M ( 0  0    I  )

               P-L ( 0  0    0  )

                    N-K-L  K   M-K  K+L-M

  ( 0 R ) =     K ( 0    R11  R12  R13  )

                M-K ( 0     0   R22  R23  )

              K+L-M ( 0     0    0   R33  )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(M) ),

  S = diag( BETA(K+1),  ... , BETA(M) ),

  C**2 + S**2 = I.

  (R11 R12 R13 ) is stored in A(1:M, N-K-L+1:N), and R33 is stored
  ( 0  R22 R23 )

  in B(M-K+1:L,N+M-K-L+1:N) on exit.
```

The routine computes C, S, R, and optionally the orthogonal transformation matrices U, V and Q.

In particular, if B is an N-by-N nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A*inv(B):

```
                      A*inv(B) = U*(D1*inv(D2))*V'.
```

If (A',B')' has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem: A'*A x = lambda* B'*B x.

In some literature, the GSVD of A and B is presented in the form U'*A*X = (0 D1), V'*B*X = (0 D2)

where U and V are orthogonal and X is nonsingular, D1 and D2 are ``diagonal''. The former GSVD form can be converted to the latter form by taking the nonsingular matrix X as

```
X = Q*( I   0    )

      ( 0 inv(R) ).
```

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Orthogonal matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Orthogonal matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Orthogonal matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in the Purpose section. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular matrix R, or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix R if M-K-L < 0. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDA >= max(1,P).
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; `ALPHA(1:K)` = 1,

  `BETA(1:K)` = 0, and if M-K-L >= 0, `ALPHA(K+1:K+L)` = C,

BETA(K+1:K+L) = S, or if M-K-L < 0, ALPHA(K+1:M) =C, ALPHA(M+1:K+L) =0

BETA(K+1:M) =S, BETA(M+1:K+L) =1 and ALPHA(K+L+1:N) = 0

BETA(K+L+1:N) = 0

- **BETA (output)**
  See the description of ALPHA.
- **U (output)**
  If JOBU = 'U', U contains the M-by-M orthogonal matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,M) if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the P-by-P orthogonal matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV >= max(1,P) if JOBV = 'V'; LDV >= 1 otherwise.
- **Q (output)**
  If JOBQ = 'Q', Q contains the N-by-N orthogonal matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= max(1,N) if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **WORK (workspace)**
  dimension (max(3*N,M,P)+N)
- **IWORK3 (output)**
  dimension(N) On exit, IWORK3 stores the sorting information. More precisely, the following loop will sort ALPHA for I = K+1, min(M,K+L) swap ALPHA(I) and ALPHA(IWORK3(I)) endfor such that ALPHA(1) >= ALPHA(2) >= ... >= ALPHA(N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = 1, the Jacobi-type procedure failed to
converge.  For further details, see subroutine STGSJA.
```

NAME

sggsvp - compute orthogonal matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

SYNOPSIS

```
 SUBROUTINE SGGSVP( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB, TOLA,
*      TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, TAU, WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER IWORK(*)
 REAL TOLA, TOLB
 REAL A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)

 SUBROUTINE SGGSVP_64( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB,
*      TOLA, TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, TAU, WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK(*)
 REAL TOLA, TOLB
 REAL A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVP( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B, [LDB],
*       TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK], [TAU],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE GGSVP_64( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B,
*       [LDB], TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK],
*       [TAU], [WORK], [INFO])
```

```
  CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
  INTEGER(8) :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  REAL :: TOLA, TOLB
  REAL, DIMENSION(:) :: TAU, WORK
  REAL, DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void sggsvp(char jobu, char jobv, char jobq, int m, int p, int n, float *a, int lda, float *b, int ldb, float tola, float tolb, int *k, int *l, float *u, int ldu, float *v, int ldv, float *q, int ldq, int *info);

void sggsvp_64(char jobu, char jobv, char jobq, long m, long p, long n, float *a, long lda, float *b, long ldb, float tola, float tolb, long *k, long *l, float *u, long ldu, float *v, long ldv, float *q, long ldq, long *info);

PURPOSE

sggsvp computes orthogonal matrices U, V and Q such that L (0 0 A23)

```
          M-K-L ( 0     0    0  )

                  N-K-L  K    L

        =     K ( 0    A12  A13 )  if M-K-L < 0;

            M-K ( 0     0   A23 )

                N-K-L  K    L

 V'*B*Q =   L ( 0     0   B13 )

          P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal. K+L = the effective numerical rank of the (M+P)-by-N matrix (A',B')'. Z' denotes the transpose of Z.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine SGGSVD.

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Orthogonal matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Orthogonal matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Orthogonal matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular (or trapezoidal) matrix described in the Purpose section.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix described in the Purpose section.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the thresholds to determine the effective numerical rank of matrix B and a subblock of A. Generally, they are set to TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS. The size of TOLA and TOLB may affect the size of backward errors of the decomposition.
- **TOLB (input)**
  See the description of TOLA.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **U (output)**
  If JOBU = 'U', U contains the orthogonal matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the orthogonal matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**

The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (output)**
  If JOBQ = 'Q', Q contains the orthogonal matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **IWORK (workspace)**
  `dimension(N)`
- **TAU (workspace)**
  `dimension(N)`
- **WORK (workspace)**
  `dimension(MAX(3*N,M,P))`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The subroutine uses LAPACK subroutine SGEQPF for the QR factorization with column pivoting to detect the effective numerical rank of the a matrix. It may be replaced by a better rank determination strategy.

NAME

sgssco - General sparse solver condition number estimate.

SYNOPSIS

```
SUBROUTINE SGSSCO ( COND, HANDLE, IER )

INTEGER           IER
REAL              COND
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSCO - Condition number estimate.

PARAMETERS

COND - REAL

On exit, an estimate of the condition number of the factored matrix. Must be called after the numerical factorization subroutine, SGSSFA().

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-700 : Invalid calling sequence - need to call SGSSFA first.
-710 : Condition number estimate not available (not implemented
       for this HANDLE's matix type).
```

NAME

sgssda - Deallocate working storage for the general sparse solver.

SYNOPSIS

```
SUBROUTINE SGSSDA ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSDA - Deallocate dynamically allocated working storage.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

none

NAME

sgssfa - General sparse solver numeric factorization.

SYNOPSIS

```
SUBROUTINE SGSSFA ( NEQNS, COLSTR, ROWIND, VALUES, HANDLE, IER )

INTEGER           NEQNS, COLSTR(*), ROWIND(*), IER
REAL              VALUES(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSFA - Numeric factorization of a sparse matrix.

PARAMETERS

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

`COLSTR(*)` - INTEGER array

On entry, `COLSTR(*)` is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

`ROWIND(*)` - INTEGER array

On entry, `ROWIND(*)` is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

`VALUES(*)` - REAL array

On entry, `VALUES(*)` is an array of size COLSTR(NEQNS+1)-1, containing the numeric values of the sparse matrix to be factored. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-300 : Invalid calling sequence - need to call SGSSOR first.
-301 : Failure to dynamically allocate memory.
-666 : Internal error.
```

NAME

sgssfs - General sparse solver one call interface.

SYNOPSIS

```
SUBROUTINE SGSSFS ( MTXTYP, PIVOT , NEQNS, COLSTR, ROWIND,
                    VALUES, NRHS  , RHS   , LDRHS , ORDMTHD,
                    OUTUNT, MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), NRHS, LDRHS,
                  OUTUNT, MSGLVL, IER
CHARACTER*3       ORDMTHD
REAL              VALUES(*), RHS(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSFS - General sparse solver one call interface.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'sp' or 'SP' - symmetric structure, positive-definite values
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, pivot specifies whether or not pivoting is used in the course of the numeric factorization. The valid options

are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

VALUES(*) - REAL array

On entry, VALUES(*) is an array of size COLSTR(NEQNS+1)-1, containing the non-zero numeric values of the sparse matrix to be factored. Unchanged on exit.

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

RHS(*) - REAL array

On entry, RHS(LDRHS,NRHS) contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see SGSSUO)
```

Unchanged on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array of containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros is less than NEQNS.
-201 : Failure to dynamically allocate memory.
-301 : Failure to dynamically allocate memory.
-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS
-666 : Internal error.

NAME

sgssin - Initialize the general sparse solver.

SYNOPSIS

```
SUBROUTINE SGSSIN ( MTXTYP, PIVOT, NEQNS, COLSTR, ROWIND, OUTUNT,
                    MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), OUTUNT, MSGLVL, IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSIN - Initialize the sparse solver and input the matrix structure.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'sp' or 'SP' - symmetric structure, positive-definite values
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, PIVOT specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
```

```
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros less than NEQNS.
```

NAME

sgssor - General sparse solver ordering and symbolic factorization.

SYNOPSIS

```
SUBROUTINE SGSSOR ( ORDMTHD, HANDLE, IER )

CHARACTER*3       ORDMTHD
INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSOR - Orders and symbolically factors a sparse matrix.

PARAMETERS

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see SGSSUO)
```

Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-200 : Invalid calling sequence - need to call SGSSIN first.
-201 : Failure to dynamically allocate memory.
-666 : Internal error.

NAME

sgssps - Print general sparse solver statics.

SYNOPSIS

```
SUBROUTINE SGSSPS ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSPS - Print solver statistics.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-800 : Invalid calling sequence - need to call SGSSSL first.
-899 : Printed solver statistics not supported this release.
```

NAME

sgssrp - Return permutation used by the general sparse solver.

SYNOPSIS

```
SUBROUTINE SGSSRP ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSRP - Returns the permutation used by the solver for the fill-reducing ordering.

PARAMETERS

PERM(NEQNS) - INTEGER array

Undefined on entry. PERM(NEQNS) is the permutation array used by the sparse solver for the fill-reducing ordering. Modified on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-600 : Invalid calling sequence - need to call SGSSOR first.
```

NAME

sgsssl - Solve routine for the general sparse solver.

SYNOPSIS

```
SUBROUTINE SGSSSL ( NRHS, RHS, LDRHS, HANDLE, IER )

INTEGER           NRHS, LDRHS, IER
REAL              RHS(LDRHS,NRHS)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSSL - Triangular solve of a factored sparse matrix.

PARAMETERS

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

`RHS(LDRHS,*)` - REAL array

On entry, `RHS(LDRHS,NRHS)` contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-400 : Invalid calling sequence - need to call SGSSFA first.
```

-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS

NAME

sgssuo - User supplied permutation for ordering used in the general sparse solver.

SYNOPSIS

```
SUBROUTINE SGSSUO ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

SGSSUO - User supplied permutation for ordering. Must be called after `SGSSIN()` (sparse solver initialization) and before `SGSSOR()` (sparse solver ordering).

PARAMETERS

`PERM(NEQNS)` - INTEGER array

On entry, `PERM(NEQNS)` is a permutation array supplied by the user for the fill-reducing ordering. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-500 : Invalid calling sequence - need to call SGSSIN first.
```

NAME

sgtcon - estimate the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF

SYNOPSIS

```
 SUBROUTINE SGTCON( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, IWORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER N, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)

 SUBROUTINE SGTCON_64( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, IWORK2, INFO)
 CHARACTER * 1 NORM
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GTCON( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK

 SUBROUTINE GTCON_64( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
 REAL :: ANORM, RCOND
```

```
REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
```

C INTERFACE

#include <sunperf.h>

void sgtcon(char norm, int n, float *low, float *diag, float *up1, float *up2, int *ipivot, float anorm, float *rcond, int *info);

void sgtcon_64(char norm, long n, float *low, float *diag, float *up1, float *up2, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

sgtcon estimates the reciprocal of the condition number of a real tridiagonal matrix A using the LU factorization as computed by SGTTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**

dimension(2*N)

- **IWORK2 (workspace)**
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

sgthr - Gathers specified elements from y into x.

SYNOPSIS

```
SUBROUTINE SGTHR(NZ, Y, X, INDX)

REAL Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE SGTHR_64(NZ, Y, X, INDX)

REAL Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHR([NZ], Y, X, INDX)

```
REAL, DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHR_64([NZ], Y, X, INDX)

REAL, DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

SGTHR - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input)

Vector in full storage form. Unchanged on exit.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

sgthrz - Gather and zero.

SYNOPSIS

```
SUBROUTINE SGTHRZ(NZ, Y, X, INDX)

REAL Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE SGTHRZ_64(NZ, Y, X, INDX)

REAL Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHRZ([NZ], Y, X, INDX)

```
REAL, DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHRZ_64([NZ], Y, X, INDX)

REAL, DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

SGTHRZ - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. The gathered elements of y are set to zero. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
  y(indx(i)) = 0
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input/output)

Vector in full storage form. Gathered elements are set to zero.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

sgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SGTRFS( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGTRFS_64( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GTRFS( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF, DIAGF,
*       UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE GTRFS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sgtrfs(char transa, int n, int nrhs, float *low, float *diag, float *up, float *lowf, float *diagf, float *upf1, float *upf2, int *ipivot, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void sgtrfs_64(char transa, long n, long nrhs, float *low, float *diag, float *up, float *lowf, float *diagf, float *upf1, float *upf2, long *ipivot, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

sgtrfs improves the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The diagonal elements of A.
- **UP (input)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.
- **DIAGF (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UPF2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 <= i <= n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SGTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sgtsv - solve the equation A*X = B,

SYNOPSIS

```
SUBROUTINE SGTSV( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
REAL LOW(*), DIAG(*), UP(*), B(LDB,*)

SUBROUTINE SGTSV_64( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
REAL LOW(*), DIAG(*), UP(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE GTSV( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: LOW, DIAG, UP
REAL, DIMENSION(:,:) :: B

SUBROUTINE GTSV_64( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: LOW, DIAG, UP
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sgtsv(int n, int nrhs, float *low, float *diag, float *up, float *b, int ldb, int *info);

void sgtsv_64(long n, long nrhs, float *low, float *diag, float *up, float *b, long ldb, long *info);

PURPOSE

sgtsv solves the equation

where A is an n by n tridiagonal matrix, by Gaussian elimination with partial pivoting.

Note that the equation A'*X = B may be solved by interchanging the order of the arguments DU and DL.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-2) elements of the second super-diagonal of the upper triangular matrix U from the LU factorization of A, in LOW(1), ..., LOW(n-2).
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of U.
- **UP (input/output)**
  On entry, UP must contain the (n-1) super-diagonal elements of A.

  On exit, UP is overwritten by the (n-1) elements of the first super-diagonal of U.
- **B (input/output)**
  On entry, the N by NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N by NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, U(i,i) is exactly zero, and the solution
has not been computed.  The factorization has not been
completed unless i  = N.
```

NAME

sgtsvx - use the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B,

---

SYNOPSIS

```
 SUBROUTINE SGTSVX( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SGTSVX_64( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GTSVX( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE GTSVX_64( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP,
*       LOWF, DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sgtsvx(char fact, char transa, int n, int nrhs, float *low, float *diag, float *up, float *lowf, float *diagf, float *upf1, float *upf2, int *ipivot, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sgtsvx_64(char fact, char transa, long n, long nrhs, float *low, float *diag, float *up, float *lowf, float *diagf, float *upf1, float *upf2, long *ipivot, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sgtsvx uses the LU factorization to compute the solution to a real system of linear equations A * X = B or A**T * X = B, where A is a tridiagonal matrix of order N and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the LU decomposition is used to factor the matrix A as A = L * U, where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

2. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': LOWF, DIAGF, UPF1, UPF2, and IPIVOT contain the factored form of A; LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2 and IPIVOT will not be modified. = 'N': The matrix will be copied to LOWF, DIAGF, and UPF1 and factored.
- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The n diagonal elements of A.
- **UP (input/output)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input/output)**
  If FACT = 'F', then LOWF is an input argument and on entry contains the (n-1) multipliers that define the matrix L from the LU factorization of A as computed by SGTTRF.

  If FACT = 'N', then LOWF is an output argument and on exit contains the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

  If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input/output)**
  If FACT = 'F', then UPF1 is an input argument and on entry contains the (n-1) elements of the first superdiagonal of U.

  If FACT = 'N', then UPF1 is an output argument and on exit contains the (n-1) elements of the first superdiagonal of U.
- **UPF2 (input/output)**
  If FACT = 'F', then UPF2 is an input argument and on entry contains the (n-2) elements of the second superdiagonal of U.

  If FACT = 'N', then UPF2 is an output argument and on exit contains the (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the LU factorization of A as computed by SGTTRF.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the LU factorization of A; row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has not been completed unless i  = N, but the
factor U is exactly singular, so the solution
and error bounds could not be computed.
RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

sgttrf - compute an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges

---

SYNOPSIS

```
SUBROUTINE SGTTRF( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
INTEGER N, INFO
INTEGER IPIVOT(*)
REAL LOW(*), DIAG(*), UP1(*), UP2(*)

SUBROUTINE SGTTRF_64( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
REAL LOW(*), DIAG(*), UP1(*), UP2(*)
```

F95 INTERFACE

```
SUBROUTINE GTTRF( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2

SUBROUTINE GTTRF_64( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2
```

C INTERFACE

#include <sunperf.h>

void sgttrf(int n, float *low, float *diag, float *up1, float *up2, int *ipivot, int *info);

void sgttrf_64(long n, float *low, float *diag, float *up1, float *up2, long *ipivot, long *info);

PURPOSE

sgttrf computes an LU factorization of a real tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

```
A = L * U
```

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

ARGUMENTS

- **N (input)**
  The order of the matrix A.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input/output)**
  On entry, UP1 must contain the (n-1) super-diagonal elements of A.

  On exit, UP1 is overwritten by the (n-1) elements of the first super-diagonal of U.
- **UP2 (output)**
  On exit, UP2 is overwritten by the (n-2) elements of the second super-diagonal of U.
- **IPIVOT (output)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -k, the k-th argument had an illegal value

 > 0:  if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

sgttrs - solve one of the systems of equations A*X = B or A'*X = B,

---

SYNOPSIS

```
 SUBROUTINE SGTTRS( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER N, NRHS, LDB, INFO
 INTEGER IPIVOT(*)
 REAL LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)

 SUBROUTINE SGTTRS_64( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 INTEGER*8 N, NRHS, LDB, INFO
 INTEGER*8 IPIVOT(*)
 REAL LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE GTTRS( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER :: N, NRHS, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2
 REAL, DIMENSION(:,:) :: B

 SUBROUTINE GTTRS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 INTEGER(8) :: N, NRHS, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: LOW, DIAG, UP1, UP2
 REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sgttrs(char transa, int n, int nrhs, float *low, float *diag, float *up1, float *up2, int *ipivot, float *b, int ldb, int *info);

void sgttrs_64(char transa, long n, long nrhs, float *low, float *diag, float *up1, float *up2, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sgttrs solves one of the systems of equations A*X = B or A'*X = B, with a tridiagonal matrix A using the LU factorization computed by SGTTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A'* X  = B  (Transpose)

  = 'C':  A'* X  = B  (Conjugate transpose  = Transpose)
  ```
- **N (input)**
  The order of the matrix A.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first super-diagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second super-diagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 <= i <= n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input/output)**
  On entry, the matrix of right hand side vectors B. On exit, B is overwritten by the solution vectors X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

shgeqz - implement a single-/double-shift version of the QZ method for finding the generalized eigenvalues w(j)=(ALPHAR(j) + i*ALPHAI(j))/BETAR(j) of the equation det(A-w(i) B) = 0 In addition, the pair A,B may be reduced to generalized Schur form

---

SYNOPSIS

```
 SUBROUTINE SHGEQZ( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)

 SUBROUTINE SHGEQZ_64( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B,
*      LDB, ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HGEQZ( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE HGEQZ_64( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void shgeqz(char job, char compq, char compz, int n, int ilo, int ihi, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *q, int ldq, float *z, int ldz, int *info);

void shgeqz_64(char job, char compq, char compz, long n, long ilo, long ihi, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *q, long ldq, float *z, long ldz, long *info);

PURPOSE

shgeqz implements a single-/double-shift version of the QZ method for finding the generalized eigenvalues B is upper triangular, and A is block upper triangular, where the diagonal blocks are either 1-by-1 or 2-by-2, the 2-by-2 blocks having complex generalized eigenvalues (see the description of the argument JOB.)

If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form by applying one orthogonal tranformation (usually called Q) on the left and another (usually called Z) on the right. The 2-by-2 upper-triangular diagonal blocks of B corresponding to 2-by-2 blocks of A will be reduced to positive diagonal matrices. (I.e., if `A(j+1,j)` is non-zero, then B(j+1,j)=B(j,j+1)=0 and `B(j,j)` and `B(j+1,j+1)` will be positive.)

If JOB='E', then at each iteration, the same transformations are computed, but they are only applied to those parts of A and B which are needed to compute ALPHAR, ALPHAI, and BETAR.

If JOB='S' and COMPQ and COMPZ are 'V' or 'I', then the orthogonal transformations used to reduce (A,B) are accumulated into the arrays Q and Z s.t.:

(in) `A(in)` Z(in)* = `Q(out)` `A(out)` Z(out)* (in) `B(in)` Z(in)* = `Q(out)` `B(out)` Z(out)*

Ref: C.B. Moler & G.W. Stewart, ``An Algorithm for Generalized Matrixigenvalue Problems'', SIAM J. Numer. Anal., 10(1973),p. 241--256.

ARGUMENTS

- **JOB (input)**

```
 = 'E': compute only ALPHAR, ALPHAI, and BETA.  A and B will
not necessarily be put into generalized Schur form.
 = 'S': put A and B into generalized Schur form, as well
as computing ALPHAR, ALPHAI, and BETA.
```

- **COMPQ (input)**

```
 = 'N': do not modify Q.

 = 'V': multiply the array Q on the right by the transpose of
the orthogonal tranformation that is applied to the
left side of A and B to reduce them to Schur form.
 = 'I': like COMPQ ='V', except that Q will be initialized to
the identity first.
```

- **COMPZ (input)**

```
 = 'N': do not modify Z.

 = 'V': multiply the array Z on the right by the orthogonal
tranformation that is applied to the right side of
A and B to reduce them to Schur form.
 = 'I': like COMPZ ='V', except that Z will be initialized to
the identity first.
```

- **N (input)**
  The order of the matrices A, B, Q, and Z. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the N-by-N upper Hessenberg matrix A. Elements below the subdiagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to generalized Schur form. If JOB ='E', then on exit A will have been destroyed. The diagonal blocks will be correct, but the off-diagonal portion will be meaningless.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. Elements below the diagonal must be zero. 2-by-2 blocks in B corresponding to 2-by-2 blocks in A will be reduced to positive diagonal form. (I.e., if `A(j+1,j)` is non-zero, then `B(j+1,j)` =B(j,j+1) =0 and `B(j,j)` and `B(j+1,j+1)` will be positive.) If JOB ='S', then on exit A and B will have been simultaneously reduced to Schur form. If JOB ='E', then on exit B will have been destroyed. Elements corresponding to diagonal blocks of A will be correct, but the off-diagonal portion will be meaningless.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **ALPHAR (output)**
  `ALPHAR(1:N)` will be set to real parts of the diagonal elements of A that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `ALPHAR(j)` =A(j,j). Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB.
- **ALPHAI (output)**
  `ALPHAI(1:N)` will be set to imaginary parts of the diagonal elements of A that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `ALPHAR(j)` =0. Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB.
- **BETA (output)**
  `BETA(1:N)` will be set to the (real) diagonal elements of B that would result from reducing A and B to Schur form and then further reducing them both to triangular form using unitary transformations s.t. the diagonal of B was non-negative real. Thus, if `A(j,j)` is in a 1-by-1 block (i.e., `A(j+1,j)` =A(j,j+1) =0), then `BETA(j)` =B(j,j). Note that the (real or complex) values (ALPHAR(j) + i*ALPHAI(j))/BETA(j), j =1,...,N, are the generalized eigenvalues of the matrix pencil A - wB. (Note that `BETA(1:N)` will always be non-negative, and no BETAI is necessary.)
- **Q (input/output)**
  If COMPQ ='N', then Q will not be referenced. If COMPQ ='V' or 'I', then the transpose of the orthogonal transformations which are applied to A and B on the left will be applied to the array Q on the right.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If COMPQ ='V' or 'I', then LDQ > = N.
- **Z (input/output)**
  If COMPZ ='N', then Z will not be referenced. If COMPZ ='V' or 'I', then the orthogonal transformations which are

applied to A and B on the right will be applied to the array Z on the right.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If COMPZ ='V' or 'I', then LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO > = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1,...,N: the QZ iteration did not converge.  (A,B) is not
in Schur form, but ALPHAR(i), ALPHAI(i), and
BETA(i), i =INFO+1,...,N should be correct.
 = N+1,...,2*N: the shift calculation failed.  (A,B) is not
in Schur form, but ALPHAR(i), ALPHAI(i), and
BETA(i), i =INFO-N+1,...,N should be correct.
 > 2*N:     various "impossible" errors.
```

FURTHER DETAILS

Iteration counters:

JITER -- counts iterations.

IITER -- counts iterations run since ILAST was last

```
          changed.  This is therefore reset only when a 1-by-1 or
          2-by-2 block deflates off the bottom.
```

NAME

shsein - use inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H

SYNOPSIS

```
 SUBROUTINE SHSEIN( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, WR, WI,
*      VL, LDVL, VR, LDVR, MM, M, WORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 INTEGER N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER IFAILL(*), IFAILR(*)
 LOGICAL SELECT(*)
 REAL H(LDH,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE SHSEIN_64( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, WR,
*      WI, VL, LDVL, VR, LDVR, MM, M, WORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 INTEGER*8 N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER*8 IFAILL(*), IFAILR(*)
 LOGICAL*8 SELECT(*)
 REAL H(LDH,*), WR(*), WI(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEIN( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], WR,
*       WI, VL, [LDVL], VR, [LDVR], MM, M, [WORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 INTEGER :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER, DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: H, VL, VR

 SUBROUTINE HSEIN_64( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], WR,
*       WI, VL, [LDVL], VR, [LDVR], MM, M, [WORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 INTEGER(8) :: N, LDH, LDVL, LDVR, MM, M, INFO
```

```
INTEGER(8), DIMENSION(:) :: IFAILL, IFAILR
LOGICAL(8), DIMENSION(:) :: SELECT
REAL, DIMENSION(:) :: WR, WI, WORK
REAL, DIMENSION(:,:) :: H, VL, VR
```

C INTERFACE

#include <sunperf.h>

void shsein(char side, char eigsrc, char initv, logical *select, int n, float *h, int ldh, float *wr, float *wi, float *vl, int ldvl, float *vr, int ldvr, int mm, int *m, int *ifaill, int *ifailr, int *info);

void shsein_64(char side, char eigsrc, char initv, logical *select, long n, float *h, long ldh, float *wr, float *wi, float *vl, long ldvl, float *vr, long ldvr, long mm, long *m, long *ifaill, long *ifailr, long *info);

PURPOSE

shsein uses inverse iteration to find specified right and/or left eigenvectors of a real upper Hessenberg matrix H.

The right eigenvector x and the left eigenvector y of the matrix H corresponding to an eigenvalue w are defined by:

```
H * x = w * x,     y**h * H = w * y**h
```

where y**h denotes the conjugate transpose of the vector y.

ARGUMENTS

- **SIDE (input)**

```
 = 'R': compute right eigenvectors only;

 = 'L': compute left eigenvectors only;

 = 'B': compute both right and left eigenvectors.
```

- **EIGSRC (input)**
  Specifies the source of eigenvalues supplied in (WR,WI):

```
 = 'Q': the eigenvalues were found using SHSEQR; thus, if
H has zero subdiagonal elements, and so is
block-triangular, then the j-th eigenvalue can be
assumed to be an eigenvalue of the block containing
the j-th row/column.  This property allows SHSEIN to
perform inverse iteration on just one diagonal block.
 = 'N': no assumptions are made on the correspondence
between eigenvalues and diagonal blocks.  In this
case, SHSEIN must always perform inverse iteration
using the whole matrix H.
```

- **INITV (input)**

= 'N': no initial vectors are supplied;

= 'U': user-supplied initial vectors are stored in the arrays
VL and/or VR.

- **SELECT (input/output)**
  Specifies the eigenvectors to be computed. To select the real eigenvector corresponding to a real eigenvalue WR(j), SELECT(j) must be set to .TRUE.. To select the complex eigenvector corresponding to a complex eigenvalue (WR(j),WI(j)), with complex conjugate (WR(j+1),WI(j+1)), either SELECT(j) or SELECT(j+1) or both must be set to .TRUE.; then on exit SELECT(j) is .TRUE. and SELECT(j+1) is .FALSE..
- **N (input)**
  The order of the matrix H. N >= 0.
- **H (input)**
  The upper Hessenberg matrix H.
- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **WR (input/output)**
  On entry, the real and imaginary parts of the eigenvalues of H; a complex conjugate pair of eigenvalues must be stored in consecutive elements of WR and WI. On exit, WR may have been altered since close eigenvalues are perturbed slightly in searching for independent eigenvectors.
- **WI (input)**
  See the description of WR.
- **VL (input/output)**
  On entry, if INITV = 'U' and SIDE = 'L' or 'B', VL must contain starting vectors for the inverse iteration for the left eigenvectors; the starting vector for each eigenvector must be in the same column(s) in which the eigenvector will be stored. On exit, if SIDE = 'L' or 'B', the left eigenvectors specified by SELECT will be stored consecutively in the columns of VL, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= max(1,N) if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if INITV = 'U' and SIDE = 'R' or 'B', VR must contain starting vectors for the inverse iteration for the right eigenvectors; the starting vector for each eigenvector must be in the same column(s) in which the eigenvector will be stored. On exit, if SIDE = 'R' or 'B', the right eigenvectors specified by SELECT will be stored consecutively in the columns of VR, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= max(1,N) if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR required to store the eigenvectors; each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  dimension((N+2)*N)
- **IFAILL (output)**
  If SIDE = 'L' or 'B', IFAILL(i) = j > 0 if the left eigenvector in the i-th column of VL (corresponding to the eigenvalue w(j)) failed to converge; IFAILL(i) = 0 if the eigenvector converged satisfactorily. If the i-th and (i+1)th columns of VL hold a complex eigenvector, then IFAILL(i) and IFAILL(i+1) are set to the same value. If SIDE = 'R', IFAILL is not referenced.
- **IFAILR (output)**

If SIDE = 'R' or 'B', IFAILR(i) = j > 0 if the right eigenvector in the i-th column of VR (corresponding to the eigenvalue w(j)) failed to converge; IFAILR(i) = 0 if the eigenvector converged satisfactorily. If the i-th and (i+1)th columns of VR hold a complex eigenvector, then IFAILR(i) and IFAILR(i+1) are set to the same value. If SIDE = 'L', IFAILR is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, i is the number of eigenvectors which
failed to converge; see IFAILL and IFAILR for further
details.
```

FURTHER DETAILS

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x|+|y|.

NAME

shseqr - compute the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors

---

SYNOPSIS

```
 SUBROUTINE SHSEQR( JOB, COMPZ, N, ILO, IHI, H, LDH, WR, WI, Z, LDZ,
*      WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 INTEGER N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL H(LDH,*), WR(*), WI(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SHSEQR_64( JOB, COMPZ, N, ILO, IHI, H, LDH, WR, WI, Z,
*      LDZ, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 INTEGER*8 N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL H(LDH,*), WR(*), WI(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEQR( JOB, COMPZ, N, ILO, IHI, H, [LDH], WR, WI, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 INTEGER :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: H, Z

 SUBROUTINE HSEQR_64( JOB, COMPZ, N, ILO, IHI, H, [LDH], WR, WI, Z,
*       [LDZ], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 INTEGER(8) :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: H, Z
```

C INTERFACE

#include <sunperf.h>

void shseqr(char job, char compz, int n, int ilo, int ihi, float *h, int ldh, float *wr, float *wi, float *z, int ldz, int *info);

void shseqr_64(char job, char compz, long n, long ilo, long ihi, float *h, long ldh, float *wr, float *wi, float *z, long ldz, long *info);

PURPOSE

shseqr computes the eigenvalues of a real upper Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**T, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q, so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q: A = Q*H*Q**T = (QZ)*T*(QZ)**T.

ARGUMENTS

- **JOB (input)**

```
= 'E':  compute eigenvalues only;

= 'S':  compute eigenvalues and the Schur form T.
```

- **COMPZ (input)**

```
= 'N':  no Schur vectors are computed;

= 'I':  Z is initialized to the unit matrix and the matrix Z
of Schur vectors of H is returned;
= 'V':  Z must contain an orthogonal matrix Q on entry, and
the product Q*Z is returned.
```

- **N (input)**
  The order of the matrix H. N > = 0.
- **ILO (input)**
  It is assumed that H is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to SGEBAL, and then passed to SGEHRD when the matrix output by SGEBAL is reduced to Hessenberg form. Otherwise ILO and IHI should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **H (input/output)**
  On entry, the upper Hessenberg matrix H. On exit, if JOB = 'S', H contains the upper quasi-triangular matrix T from the Schur decomposition (the Schur form); 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with `H(i,i)` = `H(i+1,i+1)` and `H(i+1,i)*H(i,i+1)` < 0. If JOB = 'E', the contents of H are unspecified on exit.

- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **WR (output)**
  The real and imaginary parts, respectively, of the computed eigenvalues. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of WR and WI, say the i-th and (i+1)th, with `WI(i)` > 0 and `WI(i+1)` < 0. If JOB = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in H, with `WR(i)` = `H(i,i)` and, if `H(i:i+1,i:i+1)` is a 2-by-2 diagonal block, `WI(i)` = `sqrt(H(i+1,i)*H(i,i+1))` and `WI(i+1)` = -WI(i).
- **WI (output)**
  See the description of WR.
- **Z (input/output)**
  If COMPZ = 'N': Z is not referenced.

  If COMPZ = 'I': on entry, Z need not be set, and on exit, Z contains the orthogonal matrix Z of the Schur vectors of H. If COMPZ = 'V': on entry Z must contain an N-by-N matrix Q, which is assumed to be equal to the unit matrix except for the submatrix Z(ILO:IHI,ILO:IHI); on exit Z contains Q*Z. Normally Q is the orthogonal matrix generated by SORGHR after the call to SGEHRD which formed the Hessenberg matrix H.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= `max(1,N)` if COMPZ = 'I' or 'V'; LDZ >= 1 otherwise.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, SHSEQR failed to compute all of the
eigenvalues in a total of 30*(IHI-ILO+1) iterations;
elements 1:ilo-1 and i+1:n of WR and WI contain those
eigenvalues which have been successfully computed.
```

NAME

sinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE SINQB( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE SINQB_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQB( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE SINQB_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void sinqb(int n, float *x, float *wsave);

void sinqb_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave sine synthesis of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) for scalar subroutines, initialized by SINQI.

NAME

sinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The SINQ operations are unnormalized inverses of themselves, so a call to SINQF followed by a call to SINQB will multiply the input sequence by 4 * N.

SYNOPSIS

```
SUBROUTINE SINQF( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE SINQF_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQF( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE SINQF_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void sinqf(int n, float *x, float *wsave);

void sinqf_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the quarter-wave sine transform of the input.
- **WSAVE (input)**
  On entry, an array with dimension of at least (3 * N + 15) for scalar subroutines, initialized by SINQI.

NAME

sinqi - initialize the array xWSAVE, which is used in both SINQF and SINQB.

SYNOPSIS

```
SUBROUTINE SINQI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE SINQI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE SINQI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void sinqi(int n, float *wsave);

void sinqi_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (3 * N + 15) or greater. SINQI needs to be called only once to initialize WSAVE before calling SINQF and/or SINQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

sint - compute the discrete Fourier sine transform of an odd sequence. The SINT transforms are unnormalized inverses of themselves, so a call of SINT followed by another call of SINT will multiply the input sequence by 2 * (N+1).

SYNOPSIS

```
SUBROUTINE SINT( N, X, WSAVE)
INTEGER N
REAL X(*), WSAVE(*)

SUBROUTINE SINT_64( N, X, WSAVE)
INTEGER*8 N
REAL X(*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINT( N, X, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: X, WSAVE

SUBROUTINE SINT_64( N, X, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: X, WSAVE
```

C INTERFACE

#include <sunperf.h>

void sint(int n, float *x, float *wsave);

void sint_64(long n, float *x, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N+1 is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. On exit, the sine transform of the input.
- **WSAVE (input/output)**
  On entry, an array with dimension of at least int(2.5 * N + 15) initialized by SINTI.

NAME

sinti - initialize the array WSAVE, which is used in subroutine SINT.

SYNOPSIS

```
SUBROUTINE SINTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE SINTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE SINTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void sinti(int n, float *wsave);

void sinti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2N + N/2 + 15) or greater. SINTI is called once to initialize WSAVE before calling SINT and need not be called again between calls to SINT if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

slagtf - factorize the matrix (T-lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as T-lambda*I = PLU

SYNOPSIS

```
SUBROUTINE SLAGTF( N, A, LAMBDA, B, C, TOL, D, IN, INFO)
INTEGER N, INFO
INTEGER IN(*)
REAL LAMBDA, TOL
REAL A(*), B(*), C(*), D(*)

SUBROUTINE SLAGTF_64( N, A, LAMBDA, B, C, TOL, D, IN, INFO)
INTEGER*8 N, INFO
INTEGER*8 IN(*)
REAL LAMBDA, TOL
REAL A(*), B(*), C(*), D(*)
```

F95 INTERFACE

```
SUBROUTINE LAGTF( [N], A, LAMBDA, B, C, TOL, D, IN, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IN
REAL :: LAMBDA, TOL
REAL, DIMENSION(:) :: A, B, C, D

SUBROUTINE LAGTF_64( [N], A, LAMBDA, B, C, TOL, D, IN, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IN
REAL :: LAMBDA, TOL
REAL, DIMENSION(:) :: A, B, C, D
```

C INTERFACE

#include <sunperf.h>

void slagtf(int n, float *a, float lambda, float *b, float *c, float tol, float *d, int *in, int *info);

void slagtf_64(long n, float *a, float lambda, float *b, float *c, float tol, float *d, long *in, long *info);

PURPOSE

slagtf factorizes the matrix (T - lambda*I), where T is an n by n tridiagonal matrix and lambda is a scalar, as where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

The parameter LAMBDA is included in the routine so that SLAGTF may be used, in conjunction with SLAGTS, to obtain eigenvectors of T by inverse iteration.

ARGUMENTS

- **N (input)**
  The order of the matrix T.
- **A (input/output)**
  On entry, A must contain the diagonal elements of T.

  On exit, A is overwritten by the n diagonal elements of the upper triangular matrix U of the factorization of T.
- **LAMBDA (input)**
  On entry, the scalar lambda.
- **B (input/output)**
  On entry, B must contain the (n-1) super-diagonal elements of T.

  On exit, B is overwritten by the (n-1) super-diagonal elements of the matrix U of the factorization of T.
- **C (input/output)**
  On entry, C must contain the (n-1) sub-diagonal elements of T.

  On exit, C is overwritten by the (n-1) sub-diagonal elements of the matrix L of the factorization of T.
- **TOL (input)**
  On entry, a relative tolerance used to indicate whether or not the matrix (T - lambda*I) is nearly singular. TOL should normally be chose as approximately the largest relative error in the elements of T. For example, if the elements of T are correct to about 4 significant figures, then TOL should be set to about 5*10**(-4). If TOL is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of TOL.
- **D (output)**
  On exit, D is overwritten by the (n-2) second super-diagonal elements of the matrix U of the factorization of T.
- **IN (output)**
  On exit, IN contains details of the permutation matrix P. If an interchange occurred at the kth step of the elimination, then `IN(k)` = 1, otherwise `IN(k)` = 0. The element `IN(n)` returns the smallest positive integer j such that

  abs(`u(j,j)`).le. norm((T - lambda*I)(j))*TOL,

where norm(A(j)) denotes the sum of the absolute values of the jth row of the matrix A. If no such j exists then IN(n) is returned as zero. If IN(n) is returned as positive, then a diagonal element of U is small, indicating that (T - lambda*I) is singular or nearly singular,

- **INFO (output)**

```
= 0   : successful exit
```

.lt. 0: if INFO = -k, the kth argument had an illegal value

NAME

slamrg - will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order

---

SYNOPSIS

```
SUBROUTINE SLAMRG( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER N1, N2, TRD1, TRD2
INTEGER INDEX(*)
REAL A(*)

SUBROUTINE SLAMRG_64( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER*8 N1, N2, TRD1, TRD2
INTEGER*8 INDEX(*)
REAL A(*)
```

F95 INTERFACE

```
SUBROUTINE LAMRG( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER :: N1, N2, TRD1, TRD2
INTEGER, DIMENSION(:) :: INDEX
REAL, DIMENSION(:) :: A

SUBROUTINE LAMRG_64( N1, N2, A, TRD1, TRD2, INDEX)
INTEGER(8) :: N1, N2, TRD1, TRD2
INTEGER(8), DIMENSION(:) :: INDEX
REAL, DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void slamrg(int n1, int n2, float *a, int trd1, int trd2, int *index);

void slamrg_64(long n1, long n2, float *a, long trd1, long trd2, long *index);

PURPOSE

slamrg will create a permutation list which will merge the elements of A (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

ARGUMENTS

- **N1 (input)**
  Length of the first sequence to be merged.
- **N2 (input)**
  Length of the second sequence to be merged.
- **A (input)**
  On entry, the first N1 elements of A contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final N2 elements.
- **TRD1 (input)**
  Describes the stride to be taken through the array A for the first N1 elements.

  ```
  = -1 subset is sorted in descending order.

  = 1 subset is sorted in ascending order.
  ```
- **TRD2 (input)**
  Describes the stride to be taken through the array A for the first N1 elements.

  ```
  = -1 subset is sorted in descending order.

  = 1 subset is sorted in ascending order.
  ```
- **INDEX (output)**
  On exit this array will contain a permutation such that if B(I) = A(INDEX(I)) for I =1,N1+N2, then B will be sorted in ascending order.

NAME

slarz - applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right

---

SYNOPSIS

```
SUBROUTINE SLARZ( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER M, N, L, INCV, LDC
REAL TAU
REAL V(*), C(LDC,*), WORK(*)

SUBROUTINE SLARZ_64( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER*8 M, N, L, INCV, LDC
REAL TAU
REAL V(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE LARZ( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC], [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER :: M, N, L, INCV, LDC
REAL :: TAU
REAL, DIMENSION(:) :: V, WORK
REAL, DIMENSION(:,:) :: C

SUBROUTINE LARZ_64( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER(8) :: M, N, L, INCV, LDC
REAL :: TAU
REAL, DIMENSION(:) :: V, WORK
REAL, DIMENSION(:,:) :: C
```

C INTERFACE

#include <sunperf.h>

void slarz(char side, int m, int n, int l, float *v, int incv, float tau, float *c, int ldc);

void slarz_64(char side, long m, long n, long l, float *v, long incv, float tau, float *c, long ldc);

PURPOSE

slarz applies a real elementary reflector H to a real M-by-N matrix C, from either the left or the right. H is represented in the form

```
H = I - tau * v * v'
```

where tau is a real scalar and v is a real vector.

If tau = 0, then H is taken to be the unit matrix.

H is a product of k elementary reflectors as returned by STZRZF.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form  H * C

  = 'R': form  C * H
  ```
- **M (input)**
  The number of rows of the matrix C.
- **N (input)**
  The number of columns of the matrix C.
- **L (input)**
  The number of entries of the vector V containing the meaningful part of the Householder vectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  The vector v in the representation of H as returned by STZRZF. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0.
- **TAU (input)**
  The value tau in the representation of H.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by the matrix H * C if SIDE = 'L', or C * H if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' or (M) if SIDE = 'R'

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA

NAME

slarzb - applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right

---

SYNOPSIS

```
 SUBROUTINE SLARZB( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V, LDV,
*      T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 INTEGER M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)

 SUBROUTINE SLARZB_64( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V,
*      LDV, T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 INTEGER*8 M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
```

F95 INTERFACE

```
 SUBROUTINE LARZB( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 INTEGER :: M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL, DIMENSION(:,:) :: V, T, C, WORK

 SUBROUTINE LARZB_64( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 INTEGER(8) :: M, N, K, L, LDV, LDT, LDC, LDWORK
 REAL, DIMENSION(:,:) :: V, T, C, WORK
```

C INTERFACE

#include <sunperf.h>

void slarzb(char side, char trans, char direct, char storev, int m, int n, int k, int l, float *v, int ldv, float *t, int ldt, float *c, int ldc, int ldwork);

void slarzb_64(char side, char trans, char direct, char storev, long m, long n, long k, long l, float *v, long ldv, float *t, long ldt, float *c, long ldc, long ldwork);

PURPOSE

slarzb applies a real block reflector H or its transpose H**T to a real distributed M-by-N C from the left or the right.

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **SIDE (input)**

  = 'L': apply H or H' from the Left

  = 'R': apply H or H' from the Right

- **TRANS (input)**

  = 'N': apply H (No transpose)

  = 'C': apply H' (Transpose)

- **DIRECT (input)**
  Indicates how H is formed from a product of elementary reflectors = 'F': H = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H = H(k) . . . H(2) H(1) (Backward)

- **STOREV (input)**
  Indicates how the vectors which define the elementary reflectors are stored:

  = 'C': Columnwise (not supported yet)

  = 'R': Rowwise

- **M (input)**
  The number of rows of the matrix C.

- **N (input)**
  The number of columns of the matrix C.

- **K (input)**
  The order of the matrix T (= the number of elementary reflectors whose product defines the block reflector).

- **L (input)**

The number of columns of the matrix V containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.

- **V (input)**
  If STOREV = 'C', NV = K; if STOREV = 'R', NV = L.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = L; if STOREV = 'R', LDV > = K.
- **T (input)**
  The triangular K-by-K matrix T in the representation of the block reflector.
- **LDT (input)**
  The leading dimension of the array T. LDT > = K.
- **C (output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by H*C or H'*C or C*H or C*H'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  `dimension(MAX(M,N),K)`
- **LDWORK (input)**
  The leading dimension of the array WORK. If SIDE = 'L', LDWORK > = max(1,N); if SIDE = 'R', LDWORK > = max(1,M).

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

slarzt - form the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors

---

SYNOPSIS

```
SUBROUTINE SLARZT( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
INTEGER N, K, LDV, LDT
REAL V(LDV,*), TAU(*), T(LDT,*)

SUBROUTINE SLARZT_64( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
INTEGER*8 N, K, LDV, LDT
REAL V(LDV,*), TAU(*), T(LDT,*)
```

F95 INTERFACE

```
SUBROUTINE LARZT( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
INTEGER :: N, K, LDV, LDT
REAL, DIMENSION(:) :: TAU
REAL, DIMENSION(:,:) :: V, T

SUBROUTINE LARZT_64( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
INTEGER(8) :: N, K, LDV, LDT
REAL, DIMENSION(:) :: TAU
REAL, DIMENSION(:,:) :: V, T
```

C INTERFACE

#include <sunperf.h>

void slarzt(char direct, char storev, int n, int k, float *v, int ldv, float *tau, float *t, int ldt);

void slarzt_64(char direct, char storev, long n, long k, float *v, long ldv, float *tau, float *t, long ldt);

PURPOSE

slarzt forms the triangular factor T of a real block reflector H of order > n, which is defined as a product of k elementary reflectors.

If DIRECT = 'F', H = `H(1) H(2) . . . H(k)` and T is upper triangular;

If DIRECT = 'B', H = `H(k) . . . H(2) H(1)` and T is lower triangular.

If STOREV = 'C', the vector which defines the elementary reflector `H(i)` is stored in the i-th column of the array V, and

```
H  =  I - V * T * V'
```

If STOREV = 'R', the vector which defines the elementary reflector `H(i)` is stored in the i-th row of the array V, and

```
H  =  I - V' * T * V
```

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **DIRECT (input)**
  Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

  ```
  = 'F': H  = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H  = H(k) . . . H(2) H(1) (Backward)
  ```
- **STOREV (input)**
  Specifies how the vectors which define the elementary reflectors are stored (see also Further Details):

  ```
  = 'R': rowwise
  ```
- **N (input)**
  The order of the block reflector H. N > = 0.
- **K (input)**
  The order of the triangular factor T (= the number of elementary reflectors). K > = 1.
- **V (input)**
  (LDV,K) if STOREV = 'C' (LDV,N) if STOREV = 'R' The matrix V. See further details.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = max(1,N); if STOREV = 'R', LDV > = K.

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i).
- **T (output)**
  The k by k triangular factor T of the block reflector. If DIRECT = 'F', T is upper triangular; if DIRECT = 'B', T is lower triangular. The rest of the array is not used.
- **LDT (input)**
  The leading dimension of the array T. LDT >= K.

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The shape of the matrix V and the storage of the vectors which define the H(i) is best illustrated by the following example with n = 5 and k = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

DIRECT = 'F' and STOREV = 'C': DIRECT = 'F' and STOREV = 'R':

```
                                              ______V_____

     ( v1 v2 v3 )                            /           \
     ( v1 v2 v3 )                          ( v1 v1 v1 v1 v1 . . . . 1 )
V  = ( v1 v2 v3 )                           ( v2 v2 v2 v2 v2 . . . 1   )
     ( v1 v2 v3 )                          ( v3 v3 v3 v3 v3 . . 1     )
     ( v1 v2 v3 )

        .  .  .

        .  .  .

        1  .  .

           1  .

              1
```

DIRECT = 'B' and STOREV = 'C': DIRECT = 'B' and STOREV = 'R':

```
                                                    ______V_____
        1                                          /            \
        .  1                                    ( 1 . . . . v1 v1 v1 v1 v1 )
        .  .  1                                 ( . 1 . . . v2 v2 v2 v2 v2 )
        .  .  .                                 ( . . 1 . . v3 v3 v3 v3 v3 )
        .  .  .

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

```
V  = ( v1 v2 v3 )

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

NAME

slasrt - the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D')

SYNOPSIS

```
SUBROUTINE SLASRT( ID, N, D, INFO)
CHARACTER * 1 ID
INTEGER N, INFO
REAL D(*)

SUBROUTINE SLASRT_64( ID, N, D, INFO)
CHARACTER * 1 ID
INTEGER*8 N, INFO
REAL D(*)
```

F95 INTERFACE

```
SUBROUTINE LASRT( ID, [N], D, [INFO])
CHARACTER(LEN=1) :: ID
INTEGER :: N, INFO
REAL, DIMENSION(:) :: D

SUBROUTINE LASRT_64( ID, [N], D, [INFO])
CHARACTER(LEN=1) :: ID
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: D
```

C INTERFACE

#include <sunperf.h>

void slasrt(char id, int n, float *d, int *info);

void slasrt_64(char id, long n, float *d, long *info);

PURPOSE

slasrt the numbers in D in increasing order (if ID = 'I') or in decreasing order (if ID = 'D').

Use Quick Sort, reverting to Insertion sort on arrays of

size <= 20. Dimension of STACK limits N to about 2**32.

ARGUMENTS

- **ID (input)**

```
= 'I': sort D in increasing order;

= 'D': sort D in decreasing order.
```

- **N (input)**
  The length of the array D.
- **D (input/output)**
  On entry, the array to be sorted. On exit, D has been sorted into increasing order (D(1) < = ... < = `D(N)`) or into decreasing order (D(1) > = ... > = `D(N)`), depending on ID.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

slatzm - routine is deprecated and has been replaced by routine SORMRZ

SYNOPSIS

```
SUBROUTINE SLATZM( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER M, N, INCV, LDC
REAL TAU
REAL V(*), C1(LDC,*), C2(LDC,*), WORK(*)

SUBROUTINE SLATZM_64( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
INTEGER*8 M, N, INCV, LDC
REAL TAU
REAL V(*), C1(LDC,*), C2(LDC,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE LATZM( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER :: M, N, INCV, LDC
REAL :: TAU
REAL, DIMENSION(:) :: V, WORK
REAL, DIMENSION(:,:) :: C1, C2

SUBROUTINE LATZM_64( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
INTEGER(8) :: M, N, INCV, LDC
REAL :: TAU
REAL, DIMENSION(:) :: V, WORK
REAL, DIMENSION(:,:) :: C1, C2
```

C INTERFACE

#include <sunperf.h>

void slatzm(char side, int m, int n, float *v, int incv, float tau, float *c1, float *c2, int ldc);

void slatzm_64(char side, long m, long n, float *v, long incv, float tau, float *c1, float *c2, long ldc);

PURPOSE

slatzm routine is deprecated and has been replaced by routine SORMRZ.

SLATZM applies a Householder matrix generated by STZRQF to a matrix.

Let P = I - tau*u*u', u = (1),

```
                                ( v )
```

where v is an (m-1) vector if SIDE = 'L', or a (n-1) vector if SIDE = 'R'.

If SIDE equals 'L', let

```
       C = [ C1 ] 1
           [ C2 ] m-1
             n
```

Then C is overwritten by P*C.

If SIDE equals 'R', let

```
       C = [ C1, C2 ] m
             1  n-1
```

Then C is overwritten by C*P.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form P * C

  = 'R': form C * P
  ```

- **M (input)**
  The number of rows of the matrix C.
- **N (input)**

The number of columns of the matrix C.

- **V (input)**
  (1 + (M-1)*abs(INCV)) if SIDE = 'L' (1 + (N-1)*abs(INCV)) if SIDE = 'R' The vector v in the representation of P. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0
- **TAU (input)**
  The value tau in the representation of P.
- **C1 (input/output)**
  (LDC,N) if SIDE = 'L' (M,1) if SIDE = 'R' On entry, the n-vector C1 if SIDE = 'L', or the m-vector C1 if SIDE = 'R'.

  On exit, the first row of P*C if SIDE = 'L', or the first column of C*P if SIDE = 'R'.
- **C2 (input/output)**
  (LDC, N) if SIDE = 'L' (LDC, N-1) if SIDE = 'R' On entry, the (m - 1) x n matrix C2 if SIDE = 'L', or the m x (n - 1) matrix C2 if SIDE = 'R'.

  On exit, rows 2:m of P*C if SIDE = 'L', or columns 2:m of C*P if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the arrays C1 and C2. LDC > = (1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'

NAME

snrm2 - Return the Euclidian norm of a vector.

SYNOPSIS

```
REAL FUNCTION SNRM2( N, X, INCX)
INTEGER N, INCX
REAL X(*)

REAL FUNCTION SNRM2_64( N, X, INCX)
INTEGER*8 N, INCX
REAL X(*)
```

F95 INTERFACE

```
REAL FUNCTION NRM2( [N], X, [INCX])
INTEGER :: N, INCX
REAL, DIMENSION(:) :: X

REAL FUNCTION NRM2_64( [N], X, [INCX])
INTEGER(8) :: N, INCX
REAL, DIMENSION(:) :: X
```

C INTERFACE

#include <sunperf.h>

float snrm2(int n, float *x, int incx);

float snrm2_64(long n, float *x, long incx);

PURPOSE

snrm2 Return the Euclidian norm of a vector x where x is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must be positive. Unchanged on exit.

NAME

sopgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by SSPTRD using packed storage

---

SYNOPSIS

```
SUBROUTINE SOPGTR( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDQ, INFO
REAL AP(*), TAU(*), Q(LDQ,*), WORK(*)

SUBROUTINE SOPGTR_64( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDQ, INFO
REAL AP(*), TAU(*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE OPGTR( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDQ, INFO
REAL, DIMENSION(:) :: AP, TAU, WORK
REAL, DIMENSION(:,:) :: Q

SUBROUTINE OPGTR_64( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDQ, INFO
REAL, DIMENSION(:) :: AP, TAU, WORK
REAL, DIMENSION(:,:) :: Q
```

C INTERFACE

#include <sunperf.h>

void sopgtr(char uplo, int n, float *ap, float *tau, float *q, int ldq, int *info);

void sopgtr_64(char uplo, long n, float *ap, float *tau, float *q, long ldq, long *info);

PURPOSE

sopgtr generates a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by SSPTRD using packed storage:

if UPLO = 'U', Q = `H(n-1)` . . . `H(2)` H(1),

if UPLO = 'L', Q = `H(1)` `H(2)` . . . H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangular packed storage used in previous
call to SSPTRD;
 = 'L': Lower triangular packed storage used in previous
call to SSPTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **AP (input)**
  The vectors which define the elementary reflectors, as returned by SSPTRD.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSPTRD.
- **Q (output)**
  The N-by-N orthogonal matrix Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **WORK (workspace)**
  `dimension(N-1)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sopmtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SOPMTR( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER M, N, LDC, INFO
 REAL AP(*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SOPMTR_64( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER*8 M, N, LDC, INFO
 REAL AP(*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE OPMTR( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER :: M, N, LDC, INFO
 REAL, DIMENSION(:) :: AP, TAU, WORK
 REAL, DIMENSION(:,:) :: C

 SUBROUTINE OPMTR_64( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER(8) :: M, N, LDC, INFO
 REAL, DIMENSION(:) :: AP, TAU, WORK
 REAL, DIMENSION(:,:) :: C
```

C INTERFACE

#include <sunperf.h>

void sopmtr(char side, char uplo, char trans, int m, int n, float *ap, float *tau, float *c, int ldc, int *info);

void sopmtr_64(char side, char uplo, char trans, long m, long n, float *ap, float *tau, float *c, long ldc, long *info);

PURPOSE

sopmtr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by SSPTRD using packed storage:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
```

- **UPLO (input)**

```
  = 'U': Upper triangular packed storage used in previous
call to SSPTRD;
  = 'L': Lower triangular packed storage used in previous
call to SSPTRD.
```

- **TRANS (input)**

```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **AP (input)**
  (M*(M+1)/2) if SIDE = 'L' (N*(N+1)/2) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SSPTRD. AP is modified by the routine but restored on exit.
- **TAU (input)**
  or (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSPTRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sorg2l - generate an m by n real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE SORG2L( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORG2L_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORG2L( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORG2L_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorg2l(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorg2l_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorg2l L generates an m by n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorg2r - generate an m by n real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE SORG2R( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORG2R_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORG2R( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORG2R_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorg2r(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorg2r_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorg2r R generates an m by n real matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgbr - generate one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form

---

SYNOPSIS

```
SUBROUTINE SORGBR( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
INTEGER M, N, K, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGBR_64( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
INTEGER*8 M, N, K, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGBR( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
INTEGER :: M, N, K, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGBR_64( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgbr(char vect, int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgbr_64(char vect, long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgbr generates one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form: A = Q * B * P**T. Q and P**T are defined as products of elementary reflectors `H(i)` or `G(i)` respectively.

If VECT = 'Q', A is assumed to have been an M-by-K matrix, and Q is of order M:

if m >= k, Q = `H(1) H(2) . . . H(k)` and SORGBR returns the first n columns of Q, where m >= n >= k;

if m < k, Q = `H(1) H(2) . . . H(m-1)` and SORGBR returns Q as an M-by-M matrix.

If VECT = 'P', A is assumed to have been a K-by-N matrix, and P**T is of order N:

if k < n, P**T = `G(k) . . . G(2) G(1)` and SORGBR returns the first m rows of P**T, where n >= m >= k;

if k >= n, P**T = `G(n-1) . . . G(2) G(1)` and SORGBR returns P**T as an N-by-N matrix.

ARGUMENTS

- **VECT (input)**
  Specifies whether the matrix Q or the matrix P**T is required, as defined in the transformation applied by SGEBRD:

  ```
  = 'Q':  generate Q;

  = 'P':  generate P**T.
  ```
- **M (input)**
  The number of rows of the matrix Q or P**T to be returned. M > = 0.
- **N (input)**
  The number of columns of the matrix Q or P**T to be returned. N > = 0. If VECT = 'Q', M > = N > = min(M,K); if VECT = 'P', N > = M > = min(N,K).
- **K (input)**
  If VECT = 'Q', the number of columns in the original M-by-K matrix reduced by SGEBRD. If VECT = 'P', the number of rows in the original K-by-N matrix reduced by SGEBRD. K > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SGEBRD. On exit, the M-by-N matrix Q or P**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).

- **TAU (input)**
  (min(M,K)) if VECT = 'Q' (min(N,K)) if VECT = 'P' `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or G(i), which determines Q or P**T, as returned by SGEBRD in its array argument TAUQ or TAUP.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,min(M,N)). For optimum performance LWORK >= min(M,N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sorghr - generate a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD

SYNOPSIS

```
SUBROUTINE SORGHR( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER N, ILO, IHI, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGHR_64( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER*8 N, ILO, IHI, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGHR( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
INTEGER :: N, ILO, IHI, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGHR_64( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
INTEGER(8) :: N, ILO, IHI, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorghr(int n, int ilo, int ihi, float *a, int lda, float *tau, int *info);

void sorghr_64(long n, long ilo, long ihi, float *a, long lda, float *tau, long *info);

PURPOSE

sorghr generates a real orthogonal matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by SGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **N (input)**
  The order of the matrix Q. N > = 0.
- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of SGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SGEHRD. On exit, the N-by-N orthogonal matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEHRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = IHI-ILO. For optimum performance LWORK > = (IHI-ILO)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sorgl2 - generate an m by n real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE SORGL2( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGL2_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGL2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGL2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgl2(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgl2_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgl2 generates an m by n real matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorglq - generate an M-by-N real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE SORGLQ( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGLQ_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGLQ( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGLQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorglq(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorglq_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorglq generates an M-by-N real matrix Q with orthonormal rows, which is defined as the first M rows of a product of K elementary reflectors of order N

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgql - generate an M-by-N real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE SORGQL( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGQL_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGQL( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGQL_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgql(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgql_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgql generates an M-by-N real matrix Q with orthonormal columns, which is defined as the last N columns of a product of K elementary reflectors of order M

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by SGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. M >= N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N >= K >= 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgqr - generate an M-by-N real matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE SORGQR( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGQR_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGQR( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGQR_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgqr(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgqr_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgqr generates an M-by-N real matrix Q with orthonormal columns, which is defined as the first N columns of a product of K elementary reflectors of order M

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimum performance LDWORK > = N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgr2 - generate an m by n real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE SORGR2( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGR2_64( M, N, K, A, LDA, TAU, WORK, INFO)
INTEGER*8 M, N, K, LDA, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGR2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGR2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
INTEGER(8) :: M, N, K, LDA, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgr2(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgr2_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgr2 generates an m by n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. N > = M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M > = K > = 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgrq - generate an M-by-N real matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE SORGRQ( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGRQ_64( M, N, K, A, LDA, TAU, WORK, LDWORK, INFO)
INTEGER*8 M, N, K, LDA, LDWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGRQ( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGRQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
INTEGER(8) :: M, N, K, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgrq(int m, int n, int k, float *a, int lda, float *tau, int *info);

void sorgrq_64(long m, long n, long k, float *a, long lda, float *tau, long *info);

PURPOSE

sorgrq generates an M-by-N real matrix Q with orthonormal rows, which is defined as the last M rows of a product of K elementary reflectors of order N

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by SGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

sorgtr - generate a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD

SYNOPSIS

```
SUBROUTINE SORGTR( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE SORGTR_64( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ORGTR( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE ORGTR_64( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void sorgtr(char uplo, int n, float *a, int lda, float *tau, int *info);

void sorgtr_64(char uplo, long n, float *a, long lda, float *tau, long *info);

PURPOSE

sorgtr generates a real orthogonal matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by SSYTRD:

if UPLO = 'U', Q = `H(n-1) . . . H(2)` H(1),

if UPLO = 'L', Q = `H(1) H(2) . . .` H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from SSYTRD;
 = 'L': Lower triangle of A contains elementary reflectors
from SSYTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by SSYTRD. On exit, the N-by-N orthogonal matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSYTRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N-1). For optimum performance LWORK > = (N-1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormbr - VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMBR( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMBR_64( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMBR( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMBR_64( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormbr(char vect, char side, char trans, int m, int n, int k, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormbr_64(char vect, char side, char trans, long m, long n, long k, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormbr VECT = 'Q', SORMBR overwrites the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N': Q * C C * Q TRANS = 'T': Q**T * C C * Q**T

If VECT = 'P', SORMBR overwrites the general real M-by-N matrix C with

```
                SIDE = 'L'     SIDE = 'R'
```

TRANS = 'N': P * C C * P

TRANS = 'T': P**T * C C * P**T

Here Q and P**T are the orthogonal matrices determined by SGEBRD when reducing a real matrix A to bidiagonal form: A = Q * B * P**T. Q and P**T are defined as products of elementary reflectors `H(i)` and `G(i)` respectively.

Let nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Thus nq is the order of the orthogonal matrix Q or P**T that is applied.

If VECT = 'Q', A is assumed to have been an NQ-by-K matrix: if nq >= k, Q = `H(1)` `H(2)` . . . H(k);

if nq < k, Q = `H(1)` `H(2)` . . . H(nq-1).

If VECT = 'P', A is assumed to have been a K-by-NQ matrix: if k < nq, P = `G(1)` `G(2)` . . . G(k);

if k >= nq, P = `G(1)` `G(2)` . . . G(nq-1).

ARGUMENTS

- **VECT (input)**

  ```
  = 'Q': apply Q or Q**T;

  = 'P': apply P or P**T.
  ```

- **SIDE (input)**

  ```
  = 'L': apply Q, Q**T, P or P**T from the Left;

  = 'R': apply Q, Q**T, P or P**T from the Right.
  ```

- **TRANS (input)**

```
= 'N':  No transpose, apply Q  or P;

= 'T':  Transpose, apply Q**T or P**T.
```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  If VECT = 'Q', the number of columns in the original matrix reduced by SGEBRD. If VECT = 'P', the number of rows in the original matrix reduced by SGEBRD. K >= 0.
- **A (input)**
  (LDA,min(nq,K)) if VECT = 'Q' (LDA,nq) if VECT = 'P' The vectors which define the elementary reflectors `H(i)` and G(i), whose products determine the matrices Q and P, as returned by SGEBRD.
- **LDA (input)**
  The leading dimension of the array A. If VECT = 'Q', LDA >= max(1,nq); if VECT = 'P', LDA >= max(1,min(nq,K)).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or `G(i)` which determines Q or P, as returned by SGEBRD in the array argument TAUQ or TAUP.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q or P*C or P**T*C or C*P or C*P**T.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormhr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMHR( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMHR_64( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMHR( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMHR_64( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormhr(char side, char trans, int m, int n, int ilo, int ihi, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormhr_64(char side, char trans, long m, long n, long ilo, long ihi, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormhr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of IHI-ILO elementary reflectors, as returned by SGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of SGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). If SIDE = 'L', then 1 < = ILO < = IHI < = M, if M > 0, and ILO = 1 and IHI = 0, if M = 0; if SIDE = 'R', then 1 < = ILO < = IHI < = N, if N > 0, and ILO = 1 and IHI = 0, if N = 0.

- **IHI (input)**
  See the description of ILO.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SGEHRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**

(M-1) if SIDE = 'L' (N-1) if SIDE = 'R' TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGEHRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormlq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMLQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMLQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMLQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMLQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormlq(char side, char trans, int m, int n, int k, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormlq_64(char side, char trans, long m, long n, long k, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormlq overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by SGELQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).
- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGELQF.
- **C (input/output)**

On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormql - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMQL( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMQL_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMQL( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMQL_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormql(char side, char trans, int m, int n, int k, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormql_64(char side, char trans, long m, long n, long k, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormql overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by SGEQLF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQLF in the last k columns of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).
- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGEQLF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormqr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMQR( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMQR_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMQR( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMQR_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormqr(char side, char trans, int m, int n, int k, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormqr_64(char side, char trans, long m, long n, long k, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormqr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by SGEQRF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGEQRF in the first k columns of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SGEQRF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormrq - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMRQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMRQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMRQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMRQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormrq(char side, char trans, int m, int n, int k, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormrq_64(char side, char trans, long m, long n, long k, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormrq overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by SGERQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by SGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).
- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by SGERQF.
- **C (input/output)**

On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.

- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sormrz - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER M, N, K, L, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 INTEGER*8 M, N, K, L, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMRZ( SIDE, TRANS, [M], [N], K, L, A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER :: M, N, K, L, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMRZ_64( SIDE, TRANS, [M], [N], K, L, A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 INTEGER(8) :: M, N, K, L, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormrz(char side, char trans, int m, int n, int k, int l, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormrz_64(char side, char trans, long m, long n, long k, long l, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormrz overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by STZRZF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **L (input)**
  The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by STZRZF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by STZRZF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

sormtr - overwrite the general real M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE SORMTR( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER M, N, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)

 SUBROUTINE SORMTR_64( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 INTEGER*8 M, N, LDA, LDC, LWORK, INFO
 REAL A(LDA,*), TAU(*), C(LDC,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE ORMTR( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER :: M, N, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE ORMTR_64( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 INTEGER(8) :: M, N, LDA, LDC, LWORK, INFO
 REAL, DIMENSION(:) :: TAU, WORK
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void sormtr(char side, char uplo, char trans, int m, int n, float *a, int lda, float *tau, float *c, int ldc, int *info);

void sormtr_64(char side, char uplo, char trans, long m, long n, float *a, long lda, float *tau, float *c, long ldc, long *info);

PURPOSE

sormtr overwrites the general real M-by-N matrix C with TRANS = 'T': Q**T * C C * Q**T

where Q is a real orthogonal matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by SSYTRD:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**T from the Left;

  = 'R': apply Q or Q**T from the Right.
  ```

- **UPLO (input)**

  ```
  = 'U': Upper triangle of A contains elementary reflectors
from SSYTRD;
  = 'L': Lower triangle of A contains elementary reflectors
from SSYTRD.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'T':  Transpose, apply Q**T.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by SSYTRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**

(M-1) if SIDE = 'L' (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by SSYTRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**T*C or C*Q**T or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

spbcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

SYNOPSIS

```
 SUBROUTINE SPBCON( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INFO
 INTEGER WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE SPBCON_64( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INFO
 INTEGER*8 WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBCON( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE PBCON_64( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
```

```
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spbcon(char uplo, int n, int ndiag, float *a, int lda, float anorm, float *rcond, int *info);

void spbcon_64(char uplo, long n, long ndiag, float *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

spbcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite band matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  `= 'U': Upper triangular factor stored in A;`

  `= 'L': Lower triangular factor stored in A.`
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric band matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**

dimension(N)

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

spbequ - compute row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
 SUBROUTINE SPBEQU( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INFO
 REAL SCOND, AMAX
 REAL A(LDA,*), SCALE(*)

 SUBROUTINE SPBEQU_64( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INFO
 REAL SCOND, AMAX
 REAL A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE PBEQU( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INFO
 REAL :: SCOND, AMAX
 REAL, DIMENSION(:) :: SCALE
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE PBEQU_64( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INFO
 REAL :: SCOND, AMAX
 REAL, DIMENSION(:) :: SCALE
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spbequ(char uplo, int n, int ndiag, float *a, int lda, float *scale, float *scond, float *amax, int *info);

void spbequ_64(char uplo, long n, long ndiag, float *a, long lda, float *scale, float *scond, float *amax, long *info);

PURPOSE

spbequ computes row and column scalings intended to equilibrate a symmetric positive definite band matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular of A is stored;

= 'L':  Lower triangular of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

> 0: if INFO = i, the i-th diagonal element is nonpositive.

NAME

spbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SPBRFS( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPBRFS_64( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBRFS( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE PBRFS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void spbrfs(char uplo, int n, int ndiag, int nrhs, float *a, int lda, float *af, int ldaf, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void spbrfs_64(char uplo, long n, long ndiag, long nrhs, float *a, long lda, float *af, long ldaf, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

spbrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A as computed by SPBTRF, in the same storage format as A (see A).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPBTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

spbstf - compute a split Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE SPBSTF( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER N, KD, LDAB, INFO
REAL AB(LDAB,*)

SUBROUTINE SPBSTF_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, KD, LDAB, INFO
REAL AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE PBSTF( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, KD, LDAB, INFO
REAL, DIMENSION(:,:) :: AB

SUBROUTINE PBSTF_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, KD, LDAB, INFO
REAL, DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void spbstf(char uplo, int n, int kd, float *ab, int ldab, int *info);

void spbstf_64(char uplo, long n, long kd, float *ab, long ldab, long *info);

PURPOSE

spbstf computes a split Cholesky factorization of a real symmetric positive definite band matrix A.

This routine is designed to be used in conjunction with SSBGST.

The factorization has the form A = S**T*S where S is a band matrix of the same bandwidth as A and the following structure:

```
S = ( U    )

    ( M  L )
```

where U is upper triangular of order m = (n+kd)/2, and L is lower triangular of order n-m.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the factor S from the split Cholesky factorization A = S**T*S. See Further Details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD+1.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the factorization could not be completed,
because the updated element a(i,i) was negative; the
matrix A is not positive definite.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 7, KD = 2:

S = (s11 s12 s13)

```
    (       s22  s23  s24                )

    (            s33  s34                )

    (                 s44                )

    (            s53  s54  s55           )

    (                 s64  s65  s66      )

    (                      s75  s76  s77 )
```

If UPLO = 'U', the array AB holds:

on entry: on exit:

```
 *    *   a13  a24  a35  a46  a57   *    *   s13  s24  s53  s64  s75
 *   a12  a23  a34  a45  a56  a67   *   s12  s23  s34  s54  s65  s76
a11  a22  a33  a44  a55  a66  a77  s11  s22  s33  s44  s55  s66  s77
```

If UPLO = 'L', the array AB holds:

on entry: on exit:

a11 a22 a33 a44 a55 a66 a77 s11 s22 s33 s44 s55 s66 s77 a21 a32 a43 a54 a65 a76 * s12 s23 s34 s54 s65 s76 * a31 a42 a53 a64 a64 * * s13 s24 s53 s64 s75 * *

Array elements marked * are not used by the routine.

NAME

spbsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE SPBSV( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SPBSV_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PBSV( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE PBSV_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void spbsv(char uplo, int n, int ndiag, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void spbsv_64(char uplo, long n, long ndiag, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

spbsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite band matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T * U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular band matrix, and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

spbsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SPBSVX( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPBSVX_64( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PBSVX( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

```
 SUBROUTINE PBSVX_64( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void spbsvx(char fact, char uplo, int n, int ndiag, int nrhs, float *a, int lda, float *af, int ldaf, char equed, float *scale, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void spbsvx_64(char fact, char uplo, long n, long ndiag, long nrhs, float *a, long lda, float *af, long ldaf, char equed, float *scale, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

spbsvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite band matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T * U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular band matrix, and L is a lower
triangular band matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
   equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```
- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right-hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A (see A). If EQUED = 'Y', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'Y':  Equilibration was done, i.e., A has been replaced by
diag(SCALE) * A * diag(SCALE).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11  a12  a13
        a22  a23  a24
             a33  a34  a35
                  a44  a45  a46
                       a55  a56
   (aij =conjg(aji))         a66
```

Band storage of the upper triangle of A:

```
    *    *   a13  a24  a35  a46
    *   a12  a23  a34  a45  a56
   a11  a22  a33  a44  a55  a66
```

Similarly, if UPLO = 'L' the format of A is as follows:

```
   a11  a22  a33  a44  a55  a66
   a21  a32  a43  a54  a65   *
   a31  a42  a53  a64   *    *
```

Array elements marked * are not used by the routine.

NAME

spbtf2 - compute the Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE SPBTF2( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER N, KD, LDAB, INFO
REAL AB(LDAB,*)

SUBROUTINE SPBTF2_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, KD, LDAB, INFO
REAL AB(LDAB,*)
```

F95 INTERFACE

```
SUBROUTINE PBTF2( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, KD, LDAB, INFO
REAL, DIMENSION(:,:) :: AB

SUBROUTINE PBTF2_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, KD, LDAB, INFO
REAL, DIMENSION(:,:) :: AB
```

C INTERFACE

#include <sunperf.h>

void spbtf2(char uplo, int n, int kd, float *ab, int ldab, int *info);

void spbtf2_64(char uplo, long n, long kd, float *ab, long ldab, long *info);

PURPOSE

spbtf2 computes the Cholesky factorization of a real symmetric positive definite band matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix, U' is the transpose of U, and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **KD (input)**
  The number of super-diagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. KD >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the band matrix A, in the same storage format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, KD = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

spbtrf - compute the Cholesky factorization of a real symmetric positive definite band matrix A

SYNOPSIS

```
SUBROUTINE SPBTRF( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, LDA, INFO
REAL A(LDA,*)

SUBROUTINE SPBTRF_64( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE PBTRF( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE PBTRF_64( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spbtrf(char uplo, int n, int ndiag, float *a, int lda, int *info);

void spbtrf_64(char uplo, long n, long ndiag, float *a, long lda, long *info);

PURPOSE

spbtrf computes the Cholesky factorization of a real symmetric positive definite band matrix A.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

Contributed by

Peter Mayes and Giuseppe Radicati, IBM ECSEC, Rome, March 23, 1989

NAME

spbtrs - solve a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF

SYNOPSIS

```
SUBROUTINE SPBTRS( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SPBTRS_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PBTRS( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE PBTRS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void spbtrs(char uplo, int n, int ndiag, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void spbtrs_64(char uplo, long n, long ndiag, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

spbtrs solves a system of linear equations A*X = B with a symmetric positive definite band matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPBTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular factor stored in A;

= 'L':  Lower triangular factor stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

spocon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
 SUBROUTINE SPOCON( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, LDA, INFO
 INTEGER WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE SPOCON_64( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, LDA, INFO
 INTEGER*8 WORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE POCON( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE POCON_64( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL :: ANORM, RCOND
```

```
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spocon(char uplo, int n, float *a, int lda, float anorm, float *rcond, int *info);

void spocon_64(char uplo, long n, float *a, long lda, float anorm, float *rcond, long *info);

PURPOSE

spocon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

spoequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE SPOEQU( N, A, LDA, SCALE, SCOND, AMAX, INFO)
INTEGER N, LDA, INFO
REAL SCOND, AMAX
REAL A(LDA,*), SCALE(*)

SUBROUTINE SPOEQU_64( N, A, LDA, SCALE, SCOND, AMAX, INFO)
INTEGER*8 N, LDA, INFO
REAL SCOND, AMAX
REAL A(LDA,*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE POEQU( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
INTEGER :: N, LDA, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE
REAL, DIMENSION(:,:) :: A

SUBROUTINE POEQU_64( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
INTEGER(8) :: N, LDA, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: SCALE
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spoequ(int n, float *a, int lda, float *scale, float *scond, float *amax, int *info);

void spoequ_64(long n, float *a, long lda, float *scale, float *scond, float *amax, long *info);

PURPOSE

spoequ computes row and column scalings intended to equilibrate a symmetric positive definite matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The N-by-N symmetric positive definite matrix whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the i-th diagonal element is nonpositive.
```

NAME

sporfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite,

---

SYNOPSIS

```
 SUBROUTINE SPORFS( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPORFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PORFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE PORFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B,
*        [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sporfs(char uplo, int n, int nrhs, float *a, int lda, float *af, int ldaf, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void sporfs_64(char uplo, long n, long nrhs, float *a, long lda, float *af, long ldaf, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

sporfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPOTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

sposv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE SPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SPOSV_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE POSV( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE POSV_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void sposv(char uplo, int n, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void sposv_64(char uplo, long n, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

sposv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T* U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i of A is not
  positive definite, so the factorization could not be
  completed, and the solution has not been computed.
  ```

NAME

sposvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SPOSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*       SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPOSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*       SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE POSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE POSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: WORK2
REAL :: RCOND
REAL, DIMENSION(:) :: SCALE, FERR, BERR, WORK
REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void sposvx(char fact, char uplo, int n, int nrhs, float *a, int lda, float *af, int ldaf, char equed, float *scale, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sposvx_64(char fact, char uplo, long n, long nrhs, float *a, long lda, float *af, long ldaf, char equed, float *scale, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sposvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T* U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix diag(SCALE)*A*diag(SCALE).

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise,

SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

spotf2 - compute the Cholesky factorization of a real symmetric positive definite matrix A

SYNOPSIS

```
SUBROUTINE SPOTF2( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
REAL A(LDA,*)

SUBROUTINE SPOTF2_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTF2( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE POTF2_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spotf2(char uplo, int n, float *a, int lda, int *info);

void spotf2_64(char uplo, long n, float *a, long lda, long *info);

PURPOSE

spotf2 computes the Cholesky factorization of a real symmetric positive definite matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U'*U or A = L*L'.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

NAME

spotrf - compute the Cholesky factorization of a real symmetric positive definite matrix A

SYNOPSIS

```
SUBROUTINE SPOTRF( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
REAL A(LDA,*)

SUBROUTINE SPOTRF_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTRF( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE POTRF_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spotrf(char uplo, int n, float *a, int lda, int *info);

void spotrf_64(char uplo, long n, float *a, long lda, long *info);

PURPOSE

spotrf computes the Cholesky factorization of a real symmetric positive definite matrix A.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the block version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the factorization could not be
completed.
  ```

NAME

spotri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
SUBROUTINE SPOTRI( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
REAL A(LDA,*)

SUBROUTINE SPOTRI_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE POTRI( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE POTRI_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void spotri(char uplo, int n, float *a, int lda, int *info);

void spotri_64(char uplo, long n, float *a, long lda, long *info);

PURPOSE

spotri computes the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF. On exit, the upper or lower triangle of the (symmetric) inverse of A, overwriting the input factor U or L.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

spotrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF

SYNOPSIS

```
SUBROUTINE SPOTRS( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SPOTRS_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE POTRS( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE POTRS_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void spotrs(char uplo, int n, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void spotrs_64(char uplo, long n, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

spotrs solves a system of linear equations A*X = B with a symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPOTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sppcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE SPPCON( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER WORK2(*)
REAL ANORM, RCOND
REAL A(*), WORK(*)

SUBROUTINE SPPCON_64( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 WORK2(*)
REAL ANORM, RCOND
REAL A(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PPCON( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: WORK2
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: A, WORK

SUBROUTINE PPCON_64( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: WORK2
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void sppcon(char uplo, int n, float *a, float anorm, float *rcond, int *info);

void sppcon_64(char uplo, long n, float *a, float anorm, float *rcond, long *info);

PURPOSE

sppcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite packed matrix using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the symmetric matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sppequ - compute row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE SPPEQU( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
REAL SCOND, AMAX
REAL A(*), SCALE(*)

SUBROUTINE SPPEQU_64( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
REAL SCOND, AMAX
REAL A(*), SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE PPEQU( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: A, SCALE

SUBROUTINE PPEQU_64( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL :: SCOND, AMAX
REAL, DIMENSION(:) :: A, SCALE
```

C INTERFACE

#include <sunperf.h>

void sppequ(char uplo, int n, float *a, float *scale, float *scond, float *amax, int *info);

void sppequ_64(char uplo, long n, float *a, float *scale, float *scond, float *amax, long *info);

PURPOSE

sppequ computes row and column scalings intended to equilibrate a symmetric positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). S contains the scale factors, S(i)=1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)=S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the i-th diagonal element is nonpositive.
  ```

NAME

spprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SPPRFS( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPPRFS_64( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PPRFS( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX], FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE PPRFS_64( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX],
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void spprfs(char uplo, int n, int nrhs, float *a, float *af, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void spprfs_64(char uplo, long n, long nrhs, float *a, float *af, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

spprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, as computed by SPPTRF/CPPTRF, packed columnwise in a linear array in the same format as A (see A).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**

The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).

- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sppsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE SPPSV( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)

SUBROUTINE SPPSV_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PPSV( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B

SUBROUTINE PPSV_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sppsv(char uplo, int n, int nrhs, float *a, float *b, int ldb, int *info);

void sppsv_64(char uplo, long n, long nrhs, float *a, float *b, long ldb, long *info);

PURPOSE

sppsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**T* U,  if UPLO = 'U', or

A = L * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

sppsvx - use the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SPPSVX( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL RCOND
 REAL A(*), AF(*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPPSVX_64( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL RCOND
 REAL A(*), AF(*), SCALE(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PPSVX( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: A, AF, SCALE, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE PPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE,
*       B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: WORK2
REAL :: RCOND
REAL, DIMENSION(:) :: A, AF, SCALE, FERR, BERR, WORK
REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sppsvx(char fact, char uplo, int n, int nrhs, float *a, float *af, char equed, float *scale, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sppsvx_64(char fact, char uplo, long n, long nrhs, float *a, float *af, char equed, float *scale, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sppsvx uses the Cholesky factorization A = U**T*U or A = L*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**T* U, if UPLO = 'U', or

```
   A = L * L**T,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **AF (input/output)**
  (N*(N+1)/2) If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L', in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

spptrf - compute the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE SPPTRF( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
REAL A(*)

SUBROUTINE SPPTRF_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
REAL A(*)
```

F95 INTERFACE

```
SUBROUTINE PPTRF( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL, DIMENSION(:) :: A

SUBROUTINE PPTRF_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void spptrf(char uplo, int n, float *a, int *info);

void spptrf_64(char uplo, long n, float *a, long *info);

PURPOSE

spptrf computes the Cholesky factorization of a real symmetric positive definite matrix A stored in packed format.

The factorization has the form

```
A = U**T * U,  if UPLO = 'U', or

A = L  * L**T,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, in the same storage format as A.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

spptri - compute the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE SPPTRI( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
REAL A(*)

SUBROUTINE SPPTRI_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
REAL A(*)
```

F95 INTERFACE

```
SUBROUTINE PPTRI( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL, DIMENSION(:) :: A

SUBROUTINE PPTRI_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void spptri(char uplo, int n, float *a, int *info);

void spptri_64(char uplo, long n, float *a, long *info);

PURPOSE

spptri computes the inverse of a real symmetric positive definite matrix A using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangular factor is stored in A;

 = 'L':  Lower triangular factor is stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise as a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.

  On exit, the upper or lower triangle of the (symmetric) inverse of A, overwriting the input factor U or L.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

spptrs - solve a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF

SYNOPSIS

```
SUBROUTINE SPPTRS( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)

SUBROUTINE SPPTRS_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PPTRS( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B

SUBROUTINE PPTRS_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void spptrs(char uplo, int n, int nrhs, float *a, float *b, int ldb, int *info);

void spptrs_64(char uplo, long n, long nrhs, float *a, float *b, long ldb, long *info);

PURPOSE

spptrs solves a system of linear equations A*X = B with a symmetric positive definite matrix A in packed storage using the Cholesky factorization A = U**T*U or A = L*L**T computed by SPPTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**T*U or A = L*L**T, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sptcon - compute the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF

SYNOPSIS

```
SUBROUTINE SPTCON( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
INTEGER N, INFO
REAL ANORM, RCOND
REAL DIAG(*), OFFD(*), WORK(*)

SUBROUTINE SPTCON_64( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
INTEGER*8 N, INFO
REAL ANORM, RCOND
REAL DIAG(*), OFFD(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTCON( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
INTEGER :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: DIAG, OFFD, WORK

SUBROUTINE PTCON_64( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
INTEGER(8) :: N, INFO
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: DIAG, OFFD, WORK
```

C INTERFACE

#include <sunperf.h>

void sptcon(int n, float *diag, float *offd, float anorm, float *rcond, int *info);

void sptcon_64(long n, float *diag, float *offd, float anorm, float *rcond, long *info);

PURPOSE

sptcon computes the reciprocal of the condition number (in the 1-norm) of a real symmetric positive definite tridiagonal matrix using the factorization A = L*D*L**T or A = U**T*D*U computed by SPTTRF.

`Norm(inv(A))` is computed by a direct method, and the reciprocal of the condition number is computed as

```
RCOND = 1 / (ANORM * norm(inv(A))).
```

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization of A, as computed by SPTTRF.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization of A, as computed by SPTTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The method used is described in Nicholas J. Higham, ``Efficient Algorithms for Computing the Condition Number of a Tridiagonal Matrix'', SIAM J. Sci. Stat. Comput., Vol. 7, No. 1, January 1986.

NAME

spteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor

---

SYNOPSIS

```
SUBROUTINE SPTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER N, LDZ, INFO
REAL D(*), E(*), Z(LDZ,*), WORK(*)

SUBROUTINE SPTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER*8 N, LDZ, INFO
REAL D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
REAL, DIMENSION(:,:) :: Z

SUBROUTINE PTEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void spteqr(char compz, int n, float *d, float *e, float *z, int ldz, int *info);

void spteqr_64(char compz, long n, float *d, float *e, float *z, long ldz, long *info);

PURPOSE

spteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF, and then calling SBDSQR to compute the singular values of the bidiagonal factor.

This routine computes the eigenvalues of the positive definite tridiagonal matrix to high relative accuracy. This means that if the eigenvalues range over many orders of magnitude in size, then the small eigenvalues and corresponding eigenvectors will be computed more accurately than, for example, with the standard QR method.

The eigenvectors of a full or band symmetric positive definite matrix can also be found if SSYTRD, SSPTRD, or SSBTRD has been used to reduce this matrix to tridiagonal form. (The reduction to tridiagonal form, however, may preclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix, if these eigenvalues range over many orders of magnitude.)

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvectors of original symmetric
matrix also.  Array Z contains the orthogonal
matrix used to reduce the original matrix to
tridiagonal form.
 = 'I':  Compute eigenvectors of tridiagonal matrix also.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On normal exit, D contains the eigenvalues, in descending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', the orthogonal matrix used in the reduction to tridiagonal form. On exit, if COMPZ = 'V', the orthonormal eigenvectors of the original symmetric matrix; if COMPZ = 'I', the orthonormal eigenvectors of the tridiagonal matrix. If INFO > 0 on exit, Z contains the eigenvectors associated with only the stored eigenvalues. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if COMPZ = 'V' or 'I', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(4*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

 > 0: if INFO = i, and i is:
 < = N the Cholesky factorization of the matrix could
not be performed because the i-th principal minor
was not positive definite.
 > N   the SVD algorithm failed to converge;
if INFO = N+i, i off-diagonal elements of the
bidiagonal factor did not converge to zero.

NAME

sptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SPTRFS( N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB, X,
*       LDX, FERR, BERR, WORK, INFO)
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL DIAG(*), OFFD(*), DIAGF(*), OFFDF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPTRFS_64( N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB, X,
*       LDX, FERR, BERR, WORK, INFO)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL DIAG(*), OFFD(*), DIAGF(*), OFFDF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PTRFS( [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [INFO])
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: DIAG, OFFD, DIAGF, OFFDF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE PTRFS_64( [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [INFO])
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL, DIMENSION(:) :: DIAG, OFFD, DIAGF, OFFDF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sptrfs(int n, int nrhs, float *diag, float *offd, float *diagf, float *offdf, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void sptrfs_64(long n, long nrhs, float *diag, float *offd, float *diagf, float *offdf, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

sptrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.
- **OFFD (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization computed by SPTTRF.
- **OFFDF (input)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization computed by SPTTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SPTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sptsv - compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

SYNOPSIS

```
SUBROUTINE SPTSV( N, NRHS, DIAG, SUB, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
REAL DIAG(*), SUB(*), B(LDB,*)

SUBROUTINE SPTSV_64( N, NRHS, DIAG, SUB, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
REAL DIAG(*), SUB(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PTSV( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG, SUB
REAL, DIMENSION(:,:) :: B

SUBROUTINE PTSV_64( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG, SUB
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sptsv(int n, int nrhs, float *diag, float *sub, float *b, int ldb, int *info);

void sptsv_64(long n, long nrhs, float *diag, float *sub, float *b, long ldb, long *info);

PURPOSE

sptsv computes the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

A is factored as A = L*D*L**T, and the factored form of A is then used to solve the system of equations.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the factorization A = L*DIAG*L**T.
- **SUB (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A. (SUB can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U**T*DIAG*U factorization of A.)
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the solution has not been
computed.  The factorization has not been completed
unless i  = N.
```

NAME

sptsvx - use the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE SPTSVX( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, INFO)
 CHARACTER * 1 FACT
 INTEGER N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL DIAG(*), SUB(*), DIAGF(*), SUBF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SPTSVX_64( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, INFO)
 CHARACTER * 1 FACT
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 REAL RCOND
 REAL DIAG(*), SUB(*), DIAGF(*), SUBF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE PTSVX( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [INFO])
 CHARACTER(LEN=1) :: FACT
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: DIAG, SUB, DIAGF, SUBF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE PTSVX_64( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [INFO])
 CHARACTER(LEN=1) :: FACT
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL :: RCOND
 REAL, DIMENSION(:) :: DIAG, SUB, DIAGF, SUBF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sptsvx(char fact, int n, int nrhs, float *diag, float *sub, float *diagf, float *subf, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sptsvx_64(char fact, long n, long nrhs, float *diag, float *sub, float *diagf, float *subf, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sptsvx uses the factorization A = L*D*L**T to compute the solution to a real system of linear equations A*X = B, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the matrix A is factored as A = L*D*L**T, where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form

```
A = U**T*D*U.
```

2. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, DIAGF and SUBF contain the factored form of A. DIAG, SUB, DIAGF, and SUBF will not be modified. = 'N': The matrix A will be copied to DIAGF and SUBF and factored.
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.
- **SUB (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**T factorization of A. If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**T factorization of A.

- **SUBF (input/output)**
  If FACT = 'F', then SUBF is an input argument and on entry contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A. If FACT = 'N', then SUBF is an output argument and on exit contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**T factorization of A.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 of INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

spttrf - compute the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE SPTTRF( N, DIAG, OFFD, INFO)
INTEGER N, INFO
REAL DIAG(*), OFFD(*)

SUBROUTINE SPTTRF_64( N, DIAG, OFFD, INFO)
INTEGER*8 N, INFO
REAL DIAG(*), OFFD(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRF( [N], DIAG, OFFD, [INFO])
INTEGER :: N, INFO
REAL, DIMENSION(:) :: DIAG, OFFD

SUBROUTINE PTTRF_64( [N], DIAG, OFFD, [INFO])
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: DIAG, OFFD
```

C INTERFACE

#include <sunperf.h>

void spttrf(int n, float *diag, float *offd, int *info);

void spttrf_64(long n, float *diag, float *offd, long *info);

PURPOSE

sptrf computes the L*D*L' factorization of a real symmetric positive definite tridiagonal matrix A. The factorization may also be regarded as having the form A = U'*D*U.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U'*DIAG*U factorization of A.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, the leading minor of order k is not
positive definite; if k  < N, the factorization could not
be completed, while if k  = N, the factorization was
completed, but DIAG(N)  = 0.
```

NAME

spttrs - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

SYNOPSIS

```
SUBROUTINE SPTTRS( N, NRHS, DIAG, OFFD, B, LDB, INFO)
INTEGER N, NRHS, LDB, INFO
REAL DIAG(*), OFFD(*), B(LDB,*)

SUBROUTINE SPTTRS_64( N, NRHS, DIAG, OFFD, B, LDB, INFO)
INTEGER*8 N, NRHS, LDB, INFO
REAL DIAG(*), OFFD(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE PTTRS( [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG, OFFD
REAL, DIMENSION(:,:) :: B

SUBROUTINE PTTRS_64( [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: DIAG, OFFD
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void spttrs(int n, int nrhs, float *diag, float *offd, float *b, int ldb, int *info);

void spttrs_64(long n, long nrhs, float *diag, float *offd, float *b, long ldb, long *info);

PURPOSE

spttrs solves a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF. D is a diagonal matrix specified in the vector D, L is a unit bidiagonal matrix whose subdiagonal is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **N (input)**
  The order of the tridiagonal matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the factorization A = U'*DIAG*U.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -k, the k-th argument had an illegal value
```

NAME

sptts2 - solve a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF

SYNOPSIS

```
SUBROUTINE SPTTS2( N, NRHS, D, E, B, LDB)
INTEGER N, NRHS, LDB
REAL D(*), E(*), B(LDB,*)

SUBROUTINE SPTTS2_64( N, NRHS, D, E, B, LDB)
INTEGER*8 N, NRHS, LDB
REAL D(*), E(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SPTTS2( N, NRHS, D, E, B, LDB)
INTEGER :: N, NRHS, LDB
REAL, DIMENSION(:) :: D, E
REAL, DIMENSION(:,:) :: B

SUBROUTINE SPTTS2_64( N, NRHS, D, E, B, LDB)
INTEGER(8) :: N, NRHS, LDB
REAL, DIMENSION(:) :: D, E
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sptts2(int n, int nrhs, float *d, float *e, float *b, int ldb);

void sptts2_64(long n, long nrhs, float *d, float *e, float *b, long ldb);

PURPOSE

sptts2 solves a tridiagonal system of the form A * X = B using the L*D*L' factorization of A computed by SPTTRF. D is a diagonal matrix specified in the vector D, L is a unit bidiagonal matrix whose subdiagonal is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **D (input)**
  The n diagonal elements of the diagonal matrix D from the L*D*L' factorization of A.
- **E (input)**
  The (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*D*L' factorization of A. E can also be regarded as the superdiagonal of the unit bidiagonal factor U from the factorization A = U'*D*U.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

NAME

srot - Apply a Given's rotation constructed by SROTG.

SYNOPSIS

```
SUBROUTINE SROT( N, X, INCX, Y, INCY, C, S)
INTEGER N, INCX, INCY
REAL C, S
REAL X(*), Y(*)

SUBROUTINE SROT_64( N, X, INCX, Y, INCY, C, S)
INTEGER*8 N, INCX, INCY
REAL C, S
REAL X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], X, [INCX], Y, [INCY], C, S)
INTEGER :: N, INCX, INCY
REAL :: C, S
REAL, DIMENSION(:) :: X, Y

SUBROUTINE ROT_64( [N], X, [INCX], Y, [INCY], C, S)
INTEGER(8) :: N, INCX, INCY
REAL :: C, S
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void srot(int n, float *x, int incx, float *y, int incy, float c, float s);

void srot_64(long n, float *x, long incx, float *y, long incy, float c, float s);

PURPOSE

srot Apply a Given's rotation constructed by SROTG.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **C (input)**
  On entry, the C rotation value constructed by SROTG. Unchanged on exit.
- **S (input)**
  On entry, the S rotation value constructed by SROTG. Unchanged on exit.

NAME

srotg - Construct a Given's plane rotation

SYNOPSIS

```
SUBROUTINE SROTG( A, B, C, S)
REAL A, B, C, S

SUBROUTINE SROTG_64( A, B, C, S)
REAL A, B, C, S
```

F95 INTERFACE

```
SUBROUTINE ROTG( A, B, C, S)
REAL :: A, B, C, S

SUBROUTINE ROTG_64( A, B, C, S)
REAL :: A, B, C, S
```

C INTERFACE

#include <sunperf.h>

void srotg(float *a, float *b, float *c, float *s);

void srotg_64(float *a, float *b, float *c, float *s);

PURPOSE

srotg Construct a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **A (input/output)**
  On entry, A contains the entry in the first vector that corresponds to the element to be annihilated in the second vector. On exit, contains the nonzero element of the rotated vector.
- **B (input/output)**
  On entry, B contains the entry to be annihilated in the second vector. On exit, contains either S or 1/C depending on which of the input values of A and B is larger.
- **C (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.
- **S (output)**
  See the description of C.

NAME

sroti - Apply an indexed Givens rotation.

SYNOPSIS

```
SUBROUTINE SROTI(NZ, X, INDX, Y, C, S)

INTEGER NZ
INTEGER INDX(*)
REAL C, S
REAL X(*), Y(*)

SUBROUTINE SROTI_64(NZ, X, INDX, Y, C, S)

INTEGER*8 NZ
INTEGER*8 INDX(*)
REAL C, S
REAL X(*), Y(*)
```

F95 INTERFACE SUBROUTINE ROTI([NZ], X, INDX, Y, C, S)

```
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX
REAL :: C, S
REAL, DIMENSION(:) :: X, Y

SUBROUTINE ROTI_64([NZ], X, INDX, Y, C, S)

INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
REAL :: C, S
REAL, DIMENSION(:) :: X, Y
```

PURPOSE

SROTI - Applies a Givens rotation to a sparse vector x stored in compressed form and another vector y in full storage form

```
do i = 1, n
  temp = -s * x(i) + c * y(indx(i))
  x(i) = c * x(i) + s * y(indx(i))
  y(indx(i)) = temp
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values of the compressed form.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input/output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

C (input)

Scalar defining the Givens rotation

S (input)

Scalar defining the Givens rotation

NAME

srotm - Apply a Gentleman's modified Given's rotation constructed by SROTMG.

SYNOPSIS

```
SUBROUTINE SROTM( N, X, INCX, Y, INCY, PARAM)
INTEGER N, INCX, INCY
REAL X(*), Y(*), PARAM(*)

SUBROUTINE SROTM_64( N, X, INCX, Y, INCY, PARAM)
INTEGER*8 N, INCX, INCY
REAL X(*), Y(*), PARAM(*)
```

F95 INTERFACE

```
SUBROUTINE ROTM( [N], X, [INCX], Y, [INCY], PARAM)
INTEGER :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y, PARAM

SUBROUTINE ROTM_64( [N], X, [INCX], Y, [INCY], PARAM)
INTEGER(8) :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y, PARAM
```

C INTERFACE

#include <sunperf.h>

void srotm(int n, float *x, int incx, float *y, int incy, float *param);

void srotm_64(long n, float *x, long incx, float *y, long incy, float *param);

PURPOSE

srotm Apply a Given's rotation constructed by SROTMG.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, X is overwritten by the updated vector x.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **PARAM (input)**
  On entry, the rotation values constructed by SROTMG. Unchanged on exit.

NAME

srotmg - Construct a Gentleman's modified Given's plane rotation

---

SYNOPSIS

```
SUBROUTINE SROTMG( D1, D2, B1, B2, PARAM)
REAL D1, D2, B1, B2
REAL PARAM(*)

SUBROUTINE SROTMG_64( D1, D2, B1, B2, PARAM)
REAL D1, D2, B1, B2
REAL PARAM(*)
```

F95 INTERFACE

```
SUBROUTINE ROTMG( D1, D2, B1, B2, PARAM)
REAL :: D1, D2, B1, B2
REAL, DIMENSION(:) :: PARAM

SUBROUTINE ROTMG_64( D1, D2, B1, B2, PARAM)
REAL :: D1, D2, B1, B2
REAL, DIMENSION(:) :: PARAM
```

C INTERFACE

#include <sunperf.h>

void srotmg(float *d1, float *d2, float *b1, float *b2, float *param);

void srotmg_64(float *d1, float *d2, float *b1, float *b2, float *param);

PURPOSE

srotmg Construct Gentleman's modified a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **D1 (input/output)**
  On entry, the first diagonal entry in the H matrix. On exit, changed to reflect the effect of the transformation.
- **D2 (input/output)**
  On entry, the second diagonal entry in the H matrix. On exit, changed to reflect the effect of the transformation.
- **B1 (input/output)**
  On entry, the first element of the vector to which the H matrix is applied. On exit, changed to reflect the effect of the transformation.
- **B2 (input/output)**
  On entry, the second element of the vector to which the H matrix is applied. Unchanged on exit.
- **PARAM (input/output)**
  On exit, `PARAM(1)` describes the form of the rotation matrix H, and `PARAM(2..5)` contain the H matrix.

  If `PARAM(1)` = -2 then H = I and no elements of PARAM are modified.

  If `PARAM(1)` = -1 then `PARAM(2)` = h11, `PARAM(3)` = h21, `PARAM(4)` = h12, and `PARAM(5)` = h22.

  If `PARAM(1)` = 0 then h11 = h22 = 1, `PARAM(3)` = h21, and `PARAM(4)` = h12.

  If `PARAM(1)` = 1 then h12 = 1, h21 = -1, `PARAM(2)` = h11, and `PARAM(5)` = h22.

NAME

ssbev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE SSBEV( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, NDIAG, LDA, LDZ, INFO
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBEV_64( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, NDIAG, LDA, LDZ, INFO
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEV( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, NDIAG, LDA, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, Z

 SUBROUTINE SBEV_64( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, NDIAG, LDA, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void ssbev(char jobz, char uplo, int n, int ndiag, float *a, int lda, float *w, float *z, int ldz, int *info);

void ssbev_64(char jobz, char uplo, long n, long ndiag, float *a, long lda, float *w, float *z, long ldz, long *info);

PURPOSE

ssbev computes all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG + 1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  dimension(MAX(1,3*N-2))
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, the algorithm failed to converge; i
  off-diagonal elements of an intermediate tridiagonal
  form did not converge to zero.

NAME

ssbevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE SSBEVD( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL AB(LDAB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBEVD_64( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL AB(LDAB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEVD( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, Z

 SUBROUTINE SBEVD_64( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KD, LDAB, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, Z
```

C INTERFACE

#include <sunperf.h>

void ssbevd(char jobz, char uplo, int n, int kd, float *ab, int ldab, float *w, float *z, int ldz, float *work, int lwork, int *info);

void ssbevd_64(char jobz, char uplo, long n, long kd, float *ab, long ldab, float *w, float *z, long ldz, float *work, long lwork, long *info);

PURPOSE

ssbevd computes all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, AB is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows KD and KD+1 of AB, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of AB.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD + 1.

- **W (output)**

If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK must be at least 1. If JOBZ = 'N' and N > 2, LWORK must be at least 2*N. If JOBZ = 'V' and N > 2, LWORK must be at least (1 + 5*N + 2*N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array LIWORK. If JOBZ = 'N' or N <= 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 2, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

ssbevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A

SYNOPSIS

```
 SUBROUTINE SSBEVX( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ, VL,
*      VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(LDA,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBEVX_64( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ,
*      VL, VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(LDA,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBEVX( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q, [LDQ],
*       VL, VU, IL, IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, Q, Z

 SUBROUTINE SBEVX_64( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q,
*       [LDQ], VL, VU, IL, IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK],
*       [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
```

```
INTEGER(8) :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
REAL :: VL, VU, ABTOL
REAL, DIMENSION(:) :: W, WORK
REAL, DIMENSION(:,:) :: A, Q, Z
```

C INTERFACE

#include <sunperf.h>

void ssbevx(char jobz, char range, char uplo, int n, int ndiag, float *a, int lda, float *q, int ldq, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, float *z, int ldz, int *ifail, int *info);

void ssbevx_64(char jobz, char range, char uplo, long n, long ndiag, float *a, long lda, float *q, long ldq, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

ssbevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first NDIAG+1 rows of the

array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd).

On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG + 1.
- **Q (output)**
  If JOBZ = 'V', the N-by-N orthogonal matrix used in the reduction to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'V', then LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 <= NFOUND <= N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least max(1,NFOUND) columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  dimension(7*N)
- **IWORK2 (workspace)**

- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.

  > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

ssbgst - reduce a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

---

SYNOPSIS

```
 SUBROUTINE SSBGST( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*       LDX, WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)

 SUBROUTINE SSBGST_64( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*       LDX, WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGST( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        X, [LDX], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: AB, BB, X

 SUBROUTINE SBGST_64( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        X, [LDX], [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: AB, BB, X
```

C INTERFACE

#include <sunperf.h>

void ssbgst(char vect, char uplo, int n, int ka, int kb, float *ab, int ldab, float *bb, int ldbb, float *x, int ldx, int *info);

void ssbgst_64(char vect, char uplo, long n, long ka, long kb, float *ab, long ldab, float *bb, long ldbb, float *x, long ldx, long *info);

PURPOSE

ssbgst reduces a real symmetric-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y, such that C has the same bandwidth as A.

B must have been previously factorized as S**T*S by SPBSTF, using a split Cholesky factorization. A is overwritten by C = X**T*A*X, where X = S**(-1)*Q and Q is an orthogonal matrix chosen to preserve the bandwidth of A.

ARGUMENTS

- **VECT (input)**

  ```
  = 'N':  do not form the transformation matrix X;

  = 'V':  form X.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the transformed matrix X**T*A*X, stored in the same format as A.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input)**
  The banded factor S from the split Cholesky factorization of B, as returned by SPBSTF, stored in the first KB+1

rows of the array.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **X (output)**
  If VECT = 'V', the n-by-n matrix X. If VECT = 'N', the array X is not referenced.
- **LDX (input)**
  The leading dimension of the array X. LDX >= `max(1,N)` if VECT = 'V'; LDX >= 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

ssbgv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE SSBGV( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*       LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBGV_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*       Z, LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGV( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB], W,
*        Z, [LDZ], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, BB, Z

 SUBROUTINE SBGV_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        W, Z, [LDZ], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, BB, Z
```

C INTERFACE

#include <sunperf.h>

void ssbgv(char jobz, char uplo, int n, int ka, int kb, float *ab, int ldab, float *bb, int ldbb, float *w, float *z, int ldz, int *info);

void ssbgv_64(char jobz, char uplo, long n, long ka, long kb, float *ab, long ldab, float *bb, long ldbb, float *w, float *z, long ldz, long *info);

PURPOSE

ssbgv computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` =

`B(i,j)` for `max(1,j-kb)` < =i < =j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j < =i < =min(n,j+kb).

On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then SPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

NAME

ssbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE SSBGVD( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*      LDZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBGVD_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*      Z, LDZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL AB(LDAB,*), BB(LDBB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGVD( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, BB, Z

 SUBROUTINE SBGVD_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
```

```
REAL, DIMENSION(:,:) :: AB, BB, Z
```

C INTERFACE

#include <sunperf.h>

void ssbgvd(char jobz, char uplo, int n, int ka, int kb, float *ab, int ldab, float *bb, int ldbb, float *w, float *z, int ldz, float *work, int lwork, int *info);

void ssbgvd_64(char jobz, char uplo, long n, long ka, long kb, float *ab, long ldab, float *bb, long ldbb, float *w, float *z, long ldz, float *work, long lwork, long *info);

PURPOSE

ssbgvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(ka+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= 3*N. If JOBZ = 'V' and N > 1, LWORK >= 1 + 5*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if LIWORK > 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 <= N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then SPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

ssbgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE SSBGVX( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB, LDBB,
*      Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSBGVX_64( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB,
*      LDBB, Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBGVX( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, BB, Q, Z
```

```
 SUBROUTINE SBGVX_64( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: AB, BB, Q, Z
```

C INTERFACE

#include <sunperf.h>

void ssbgvx(char jobz, char range, char uplo, int n, int ka, int kb, float *ab, int ldab, float *bb, int ldbb, float *q, int ldq, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *ifail, int *info);

void ssbgvx_64(char jobz, char range, char uplo, long n, long ka, long kb, float *ab, long ldab, float *bb, long ldbb, float *q, long ldq, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

ssbgvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**

The order of the matrices A and B. N >= 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(ka+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**T*S, as returned by SPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **Q (output)**
  If JOBZ = 'V', the n-by-n matrix used in the reduction of A*x = (lambda)*B*x to standard form, i.e. C*x = (lambda)*x, and consequently C to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'N', LDQ >= 1. If JOBZ = 'V', LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 <= M <= N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so Z**T*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvalues that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0 : successful exit

 < 0 : if INFO  = -i, the i-th argument had an illegal value

 < = N: if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in IFAIL.
 > N : SPBSTF returned an error code; i.e.,
if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

ssbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE SSBMV( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 UPLO
 INTEGER N, NDIAG, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)

 SUBROUTINE SSBMV_64( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA,
*      Y, INCY)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NDIAG, LDA, INCX, INCY
 REAL ALPHA, BETA
 REAL A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
 SUBROUTINE SBMV( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX], BETA,
*       Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NDIAG, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE SBMV_64( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NDIAG, LDA, INCX, INCY
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:) :: X, Y
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssbmv(char uplo, int n, int ndiag, float alpha, float *a, int lda, float *x, int incx, float beta, float *y, int incy);

void ssbmv_64(char uplo, long n, long ndiag, float alpha, float *a, long lda, float *x, long incx, float beta, float *y, long incy);

PURPOSE

ssbmv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with ndiag super-diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the band matrix A is being supplied as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is being supplied.

  UPLO = 'L' or 'l' The lower triangular part of A is being supplied.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry, NDIAG specifies the number of super-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

  Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
   DO 20, J = 1, N
     M = 1 - J
     DO 10, I = J, MIN( N, J + NDIAG )
       A( M + I, J ) = matrix( I, J )
10   CONTINUE
20 CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ssbtrd - reduce a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
 SUBROUTINE SSBTRD( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ, WORK,
*      INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER N, KD, LDAB, LDQ, INFO
 REAL AB(LDAB,*), D(*), E(*), Q(LDQ,*), WORK(*)

 SUBROUTINE SSBTRD_64( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ,
*      WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 INTEGER*8 N, KD, LDAB, LDQ, INFO
 REAL AB(LDAB,*), D(*), E(*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SBTRD( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER :: N, KD, LDAB, LDQ, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: AB, Q

 SUBROUTINE SBTRD_64( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 INTEGER(8) :: N, KD, LDAB, LDQ, INFO
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: AB, Q
```

C INTERFACE

#include <sunperf.h>

void ssbtrd(char vect, char uplo, int n, int kd, float *ab, int ldab, float *d, float *e, float *q, int ldq, int *info);

void ssbtrd_64(char vect, char uplo, long n, long kd, float *ab, long ldab, float *d, float *e, float *q, long ldq, long *info);

PURPOSE

ssbtrd reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **VECT (input)**

  ```
  = 'N':  do not form Q;

  = 'V':  form Q;

  = 'U':  update a matrix X, by forming X*Q.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD >= 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the symmetric band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). On exit, the diagonal elements of AB are overwritten by the diagonal elements of the tridiagonal matrix T; if KD > 0, the elements on the first superdiagonal (if UPLO = 'U') or the first subdiagonal (if UPLO = 'L') are overwritten by the off-diagonal elements of T; the rest of AB is overwritten by values generated during the reduction.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.

- **D (output)**
  The diagonal elements of the tridiagonal matrix T.

- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `T(i,i+1)` if UPLO = 'U'; `E(i)` = `T(i+1,i)` if UPLO = 'L'.

- **Q (input/output)**
  On entry, if VECT = 'U', then Q must contain an N-by-N matrix X; if VECT = 'N' or 'V', then Q need not be set.

  On exit: if VECT = 'V', Q contains the N-by-N orthogonal matrix Q; if VECT = 'U', Q contains the product X*Q; if VECT = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1, and LDQ >= N if VECT = 'V' or 'U'.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Modified by Linda Kaufman, Bell Labs.

NAME

sscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE SSCAL( N, ALPHA, Y, INCY)
INTEGER N, INCY
REAL ALPHA
REAL Y(*)

SUBROUTINE SSCAL_64( N, ALPHA, Y, INCY)
INTEGER*8 N, INCY
REAL ALPHA
REAL Y(*)
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
INTEGER :: N, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: Y

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
INTEGER(8) :: N, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: Y
```

C INTERFACE

#include <sunperf.h>

void sscal(int n, float alpha, float *y, int incy);

void sscal_64(long n, float alpha, float *y, long incy);

PURPOSE

sscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

ssctr - Scatters elements from x into y.

SYNOPSIS

```
SUBROUTINE SSCTR(NZ, X, INDX, Y)

REAL X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE SSCTR_64(NZ, X, INDX, Y)

REAL X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE SCTR([NZ], X, INDX, Y)

```
REAL, DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE SCTR_64([NZ], X, INDX, Y)

REAL, DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

SSCTR - Scatters the components of a sparse vector x stored in compressed form into specified components of a vector y in full storage form.

```
do i = 1, n
  y(indx(i)) = x(i)
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values to be scattered from compressed form into full storage form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector whose elements specified by indx have been set to the corresponding entries of x. Only the elements corresponding to the indices in indx have been modified.

NAME

sspcon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
 SUBROUTINE SSPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, IWORK2,
*      INFO)
 CHARACTER * 1 UPLO
 INTEGER N, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL A(*), WORK(*)

 SUBROUTINE SSPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL A(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: A, WORK

 SUBROUTINE SPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
 REAL :: ANORM, RCOND
```

```
REAL, DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void sspcon(char uplo, int n, float *a, int *ipivot, float anorm, float *rcond, int *info);

void sspcon_64(char uplo, long n, float *a, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

sspcon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **IWORK2 (workspace)**
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sspev - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
SUBROUTINE SSPEV( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER N, LDZ, INFO
REAL A(*), W(*), Z(LDZ,*), WORK(*)

SUBROUTINE SSPEV_64( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER*8 N, LDZ, INFO
REAL A(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SPEV( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: A, W, WORK
REAL, DIMENSION(:,:) :: Z

SUBROUTINE SPEV_64( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: A, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspev(char jobz, char uplo, int n, float *a, float *w, float *z, int ldz, int *info);

void sspev_64(char jobz, char uplo, long n, float *a, float *w, float *z, long ldz, long *info);

PURPOSE

sspev computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

sspevd - compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE SSPEVD( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL AP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSPEVD_64( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK,
*      IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL AP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPEVD( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: AP, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE SPEVD_64( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: AP, W, WORK
 REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspevd(char jobz, char uplo, int n, float *ap, float *w, float *z, int ldz, float *work, int lwork, int *info);

void sspevd_64(char jobz, char uplo, long n, float *ap, float *w, float *z, long ldz, float *work, long lwork, long *info);

PURPOSE

sspevd computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, AP is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least 2*N. If JOBZ = 'V' and N > 1, LWORK must be at least 1 + 6*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N < = 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

sspevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE SSPEVX( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU, ABTOL,
*      NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSPEVX_64( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPEVX( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU, ABTOL,
*       [NFOUND], W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: A, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE SPEVX_64( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU,
*       ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
```

```
REAL, DIMENSION(:) :: A, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspevx(char jobz, char range, char uplo, int n, float *a, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, float *z, int ldz, int *ifail, int *info);

void sspevx_64(char jobz, char range, char uplo, long n, float *a, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

sspevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. Eigenvalues/vectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found;

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  If INFO = 0, the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(8*N)`
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

sspgst - reduce a real symmetric-definite generalized eigenproblem to standard form, using packed storage

---

SYNOPSIS

```
SUBROUTINE SSPGST( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, INFO
REAL AP(*), BP(*)

SUBROUTINE SSPGST_64( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, INFO
REAL AP(*), BP(*)
```

F95 INTERFACE

```
SUBROUTINE SPGST( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, INFO
REAL, DIMENSION(:) :: AP, BP

SUBROUTINE SPGST_64( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, INFO
REAL, DIMENSION(:) :: AP, BP
```

C INTERFACE

#include <sunperf.h>

void sspgst(int itype, char uplo, int n, float *ap, float *bp, int *info);

void sspgst_64(long itype, char uplo, long n, float *ap, float *bp, long *info);

PURPOSE

sspgst reduces a real symmetric-definite generalized eigenproblem to standard form, using packed storage.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**T)*A*inv(U)` or `inv(L)*A*inv(L**T)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**T or L**T*A*L.

B must have been previously factorized as U**T*U or L*L**T by SPPTRF.

ARGUMENTS

- **ITYPE (input)**

```
= 1: compute inv(U**T)*A*inv(U) or inv(L)*A*inv(L**T);

= 2 or 3: compute U*A*U**T or L**T*A*L.
```

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored and B is factored as
U**T*U;
 = 'L':  Lower triangle of A is stored and B is factored as
L*L**T.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.
- **BP (input)**
  The triangular factor from the Cholesky factorization of B, stored in the same format as A, as returned by SPPTRF.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sspgv - compute all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSPGV( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDZ, INFO
 REAL A(*), B(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSPGV_64( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDZ, INFO
 REAL A(*), B(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGV( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDZ, INFO
 REAL, DIMENSION(:) :: A, B, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE SPGV_64( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDZ, INFO
 REAL, DIMENSION(:) :: A, B, W, WORK
 REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspgv(int itype, char jobz, char uplo, int n, float *a, float *b, float *w, float *z, int ldz, int *info);

void sspgv_64(long itype, char jobz, char uplo, long n, float *a, float *b, float *w, float *z, long ldz, long *info);

PURPOSE

sspgv computes all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  (N*(N+1)/2) On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of A are destroyed.

- **B (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array B as follows: if UPLO = 'U', B(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', B(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEV returned an error code:

 < = N:  if INFO  = i, SSPEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero.
 > N:   if INFO  = n + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

sspgvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSPGVD( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSPGVD_64( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGVD( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ], [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: AP, BP, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE SPGVD_64( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ],
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: AP, BP, W, WORK
```

```
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspgvd(int itype, char jobz, char uplo, int n, float *ap, float *bp, float *w, float *z, int ldz, int *info);

void sspgvd_64(long itype, char jobz, char uplo, long n, float *ap, float *bp, float *w, float *z, long ldz, long *info);

PURPOSE

sspgvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th

column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = A(i,j) for 1 <=i <=j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = A(i,j) for j <=i <=n.

On exit, the contents of AP are destroyed.

- **BP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = B(i,j) for 1 <=i <=j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = B(i,j) for j <=i <=n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= 2*N. If JOBZ = 'V' and N > 1, LWORK >= 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEVD returned an error code:

 <= N:  if INFO  = i, SSPEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

sspgvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSPGVX( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU, IL,
*      IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSPGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU,
*      IL, IU, ABSTOL, M, W, Z, LDZ, WORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL AP(*), BP(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPGVX( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], [WORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: AP, BP, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE SPGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU,
*       IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
```

```
INTEGER(8) :: ITYPE, N, IL, IU, M, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
REAL :: VL, VU, ABSTOL
REAL, DIMENSION(:) :: AP, BP, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sspgvx(int itype, char jobz, char range, char uplo, int n, float *ap, float *bp, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *ifail, int *info);

void sspgvx_64(long itype, char jobz, char range, char uplo, long n, float *ap, float *bp, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

sspgvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric, stored in packed storage, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

= 'U': Upper triangle of A and B are stored;

= 'L': Lower triangle of A and B are stored.

- **N (input)**
  The order of the matrix pencil (A,B). N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T, in the same storage format as B.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

  If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  `dimension(8*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPPTRF or SSPEVX returned an error code:

 < = N:  if INFO  = i, SSPEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

sspmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE SSPMV( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY
REAL ALPHA, BETA
REAL A(*), X(*), Y(*)

SUBROUTINE SSPMV_64( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY
REAL ALPHA, BETA
REAL A(*), X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SPMV( UPLO, N, ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY
REAL :: ALPHA, BETA
REAL, DIMENSION(:) :: A, X, Y

SUBROUTINE SPMV_64( UPLO, N, ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY
REAL :: ALPHA, BETA
REAL, DIMENSION(:) :: A, X, Y
```

C INTERFACE

#include <sunperf.h>

void sspmv(char uplo, int n, float alpha, float *a, float *x, int incx, float beta, float *y, int incy);

void sspmv_64(char uplo, long n, float alpha, float *a, float *x, long incx, float beta, float *y, long incy);

PURPOSE

sspmv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

sspr - perform the symmetric rank 1 operation A := alpha*x*x' + A

SYNOPSIS

```
SUBROUTINE SSPR( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
INTEGER N, INCX
REAL ALPHA
REAL X(*), A(*)

SUBROUTINE SSPR_64( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX
REAL ALPHA
REAL X(*), A(*)
```

F95 INTERFACE

```
SUBROUTINE SPR( UPLO, N, ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX
REAL :: ALPHA
REAL, DIMENSION(:) :: X, A

SUBROUTINE SPR_64( UPLO, N, ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX
REAL :: ALPHA
REAL, DIMENSION(:) :: X, A
```

C INTERFACE

#include <sunperf.h>

void sspr(char uplo, int n, float alpha, float *x, int incx, float *a);

void sspr_64(char uplo, long n, float alpha, float *x, long incx, float *a);

PURPOSE

sspr performs the symmetric rank 1 operation A := alpha*x*x' + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix.

NAME

sspr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

SYNOPSIS

```
SUBROUTINE SSPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY
REAL ALPHA
REAL X(*), Y(*), A(*)

SUBROUTINE SSPR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY
REAL ALPHA
REAL X(*), Y(*), A(*)
```

F95 INTERFACE

```
SUBROUTINE SPR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y, A

SUBROUTINE SPR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y, A
```

C INTERFACE

#include <sunperf.h>

void sspr2(char uplo, int n, float alpha, float *x, int incx, float *y, int incy, float *a);

void sspr2_64(char uplo, long n, float alpha, float *x, long incx, float *y, long incy, float *a);

PURPOSE

sspr2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix.

NAME

ssprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SSPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SSPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE SPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void ssprfs(char uplo, int n, int nrhs, float *a, float *af, int *ipivot, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void ssprfs_64(char uplo, long n, long nrhs, float *a, float *af, long *ipivot, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

ssprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SSPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If

XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

sspsv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE SSPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)
REAL A(*), B(LDB,*)

SUBROUTINE SSPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
REAL A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SPSV( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B

SUBROUTINE SPSV_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sspsv(char uplo, int n, int nrhs, float *a, int *ipivot, float *b, int ldb, int *info);

void sspsv_64(char uplo, long n, long nrhs, float *a, long *ipivot, float *b, long ldb, long *info);

PURPOSE

sspsv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by SSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = aji)

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

sspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE SSPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SSPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(*), AF(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X,
*       [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE SPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
```

```
CHARACTER(LEN=1) :: FACT, UPLO
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
REAL :: RCOND
REAL, DIMENSION(:) :: A, AF, FERR, BERR, WORK
REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void sspsvx(char fact, char uplo, int n, int nrhs, float *a, float *af, int *ipivot, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void sspsvx_64(char fact, char uplo, long n, long nrhs, float *a, float *af, long *ipivot, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

sspsvx uses the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**T, if UPLO = 'U', or

```
    A = L * D * L**T,  if UPLO = 'L',

where U (or L) is a product of permutation and unit upper (lower)
triangular matrices and D is symmetric and block diagonal with
1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.
- **AF (input/output)**
  (N*(N+1)/2) If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by SSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by SSPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the

largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

ssptrd - reduce a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE SSPTRD( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
REAL AP(*), D(*), E(*), TAU(*)

SUBROUTINE SSPTRD_64( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
REAL AP(*), D(*), E(*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRD( UPLO, N, AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
REAL, DIMENSION(:) :: AP, D, E, TAU

SUBROUTINE SPTRD_64( UPLO, N, AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: AP, D, E, TAU
```

C INTERFACE

#include <sunperf.h>

void ssptrd(char uplo, int n, float *ap, float *d, float *e, float *tau, int *info);

void ssptrd_64(char uplo, long n, float *ap, float *d, float *e, float *tau, long *info);

PURPOSE

ssptrd reduces a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
   Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
   H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in AP, overwriting A(1:i-1,i+1), and tau is stored in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
   Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
   H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in AP, overwriting A(i+2:n,i), and tau is stored in TAU(i).

NAME

ssptrf - compute the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE SSPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER IPIVOT(*)
REAL A(*)

SUBROUTINE SSPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
REAL A(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A

SUBROUTINE SPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssptrf(char uplo, int n, float *a, int *ipivot, int *info);

void ssptrf_64(char uplo, long n, float *a, long *ipivot, long *info);

PURPOSE

ssptrf computes the factorization of a real symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

ssptri - compute the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
SUBROUTINE SSPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, INFO
INTEGER IPIVOT(*)
REAL A(*), WORK(*)

SUBROUTINE SSPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
REAL A(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRI( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A, WORK

SUBROUTINE SPTRI_64( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void ssptri(char uplo, int n, float *a, int *ipivot, int *info);

void ssptri_64(char uplo, long n, float *a, long *ipivot, long *info);

PURPOSE

ssptri computes the inverse of a real symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.

- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.

- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

ssptrs - solve a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF

SYNOPSIS

```
SUBROUTINE SSPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)
REAL A(*), B(LDB,*)

SUBROUTINE SSPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
REAL A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SPTRS( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B

SUBROUTINE SPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void ssptrs(char uplo, int n, int nrhs, float *a, int *ipivot, float *b, int ldb, int *info);

void ssptrs_64(char uplo, long n, long nrhs, float *a, long *ipivot, float *b, long ldb, long *info);

PURPOSE

ssptrs solves a system of linear equations A*X = B with a real symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by SSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

sstebz - compute the eigenvalues of a symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE SSTEBZ( RANGE, ORDER, N, VL, VU, IL, IU, ABSTOL, D, E, M,
*      NSPLIT, W, IBLOCK, ISPLIT, WORK, IWORK, INFO)
 CHARACTER * 1 RANGE, ORDER
 INTEGER N, IL, IU, M, NSPLIT, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), WORK(*)

 SUBROUTINE SSTEBZ_64( RANGE, ORDER, N, VL, VU, IL, IU, ABSTOL, D, E,
*      M, NSPLIT, W, IBLOCK, ISPLIT, WORK, IWORK, INFO)
 CHARACTER * 1 RANGE, ORDER
 INTEGER*8 N, IL, IU, M, NSPLIT, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEBZ( RANGE, ORDER, [N], VL, VU, IL, IU, ABSTOL, D, E,
*       M, NSPLIT, W, IBLOCK, ISPLIT, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: RANGE, ORDER
 INTEGER :: N, IL, IU, M, NSPLIT, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEBZ_64( RANGE, ORDER, [N], VL, VU, IL, IU, ABSTOL, D,
*       E, M, NSPLIT, W, IBLOCK, ISPLIT, [WORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: RANGE, ORDER
 INTEGER(8) :: N, IL, IU, M, NSPLIT, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void sstebz(char range, char order, int n, float vl, float vu, int il, int iu, float abstol, float *d, float *e, int *m, int *nsplit, float *w, int *iblock, int *isplit, int *info);

void sstebz_64(char range, char order, long n, float vl, float vu, long il, long iu, float abstol, float *d, float *e, long *m, long *nsplit, float *w, long *iblock, long *isplit, long *info);

PURPOSE

sstebz computes the eigenvalues of a symmetric tridiagonal matrix T. The user may ask for all eigenvalues, all eigenvalues in the half-open interval (VL, VU], or the IL-th through IU-th eigenvalues.

To avoid overflow, the matrix must be scaled so that its

largest element is no greater than overflow**(1/2) *

underflow**(1/4) in absolute value, and for greatest

accuracy, it should not be much smaller than that.

See W. Kahan ``Accurate Eigenvalues of a Symmetric Tridiagonal Matrix", Report CS41, Computer Science Dept., Stanford

University, July 21, 1966.

ARGUMENTS

- **RANGE (input)**

```
 = 'A': ("All")   all eigenvalues will be found.

 = 'V': ("Value") all eigenvalues in the half-open interval
(VL, VU] will be found.
 = 'I': ("Index") the IL-th through IU-th eigenvalues (of the
entire matrix) will be found.
```

- **ORDER (input)**

```
 = 'B': ("By Block") the eigenvalues will be grouped by
split-off block (see IBLOCK, ISPLIT) and
ordered from smallest to largest within
the block.
 = 'E': ("Entire matrix")
the eigenvalues for the entire matrix
will be ordered from smallest to
largest.
```

- **N (input)**

The order of the tridiagonal matrix T. N > = 0.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to VL, or greater than VU, will not be returned. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less. If ABSTOL is less than or equal to zero, then ULP*|T| will be used, where |T| means the 1-norm of T.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) off-diagonal elements of the tridiagonal matrix T.
- **M (output)**
  The actual number of eigenvalues found. 0 < = M < = N. (See also the description of INFO =2,3.)
- **NSPLIT (output)**
  The number of diagonal blocks in the matrix T. 1 < = NSPLIT < = N.
- **W (output)**
  On exit, the first M elements of W will contain the eigenvalues. (SSTEBZ may use the remaining N-M elements as workspace.)
- **IBLOCK (output)**
  At each row/column j where `E(j)` is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit, if INFO = 0, `IBLOCK(i)` specifies to which block (from 1 to the number of blocks) the eigenvalue `W(i)` belongs. (SSTEBZ may use the remaining N-M elements as workspace.)
- **ISPLIT (output)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc., and the NSPLIT-th consists of rows/columns ISPLIT(NSPLIT-1)+1 through `ISPLIT(NSPLIT)` =N. (Only the first NSPLIT elements will actually be used, but since the user cannot know a priori what value NSPLIT will have, N words must be reserved for ISPLIT.)
- **WORK (workspace)**
  `dimension(4*N)`
- **IWORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  some or all of the eigenvalues failed to converge or
```

were not computed:

```
 =1 or 3: Bisection failed to converge for some
eigenvalues; these eigenvalues are flagged by a
negative block number.  The effect is that the
eigenvalues may not be as accurate as the
absolute and relative tolerances.  This is
generally caused by unexpectedly inaccurate
arithmetic.
 =2 or 3: RANGE ='I' only: Not all of the eigenvalues
IL:IU were found.
```

Effect: M < IU+1-IL

Cause: non-monotonic arithmetic, causing the Sturm sequence to be non-monotonic. Cure: recalculate, using RANGE ='A', and pick

out eigenvalues IL:IU. = 4: RANGE ='I', and the Gershgorin interval initially used was too small. No eigenvalues were computed.

NAME

sstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

---

SYNOPSIS

```
 SUBROUTINE SSTEDC( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 COMPZ
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL D(*), E(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEDC_64( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 COMPZ
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEDC( COMPZ, N, D, E, Z, [LDZ], WORK, [LWORK], IWORK,
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEDC_64( COMPZ, N, D, E, Z, [LDZ], WORK, [LWORK], IWORK,
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: D, E, WORK
```

```
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstedc(char compz, int n, float *d, float *e, float *z, int ldz, float *work, int lwork, int *iwork, int liwork, int *info);

void sstedc_64(char compz, long n, float *d, float *e, float *z, long ldz, float *work, long lwork, long *iwork, long liwork, long *info);

PURPOSE

sstedc computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric matrix can also be found if SSYTRD or SSPTRD or SSBTRD has been used to reduce this matrix to tridiagonal form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLAED3 for details.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'I':  Compute eigenvectors of tridiagonal matrix also.

 = 'V':  Compute eigenvectors of original dense symmetric
matrix also.  On entry, Z contains the orthogonal
matrix used to reduce the original matrix to
tridiagonal form.
```

- **N (input)**
  The dimension of the symmetric tridiagonal matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the orthogonal matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original symmetric matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If eigenvectors are desired, then LDZ > = max(1,N).

- **WORK (output)**
  dimension (LWORK) On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If COMPZ = 'N' or N < = 1 then LWORK must be at least 1. If COMPZ = 'V' and N > 1 then LWORK must be at least (1 + 3*N + 2*N*lg N + 3*N**2), where lg(N) = smallest integer k such that 2**k > = N. If COMPZ = 'I' and N > 1 then LWORK must be at least (1 + 4*N + N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (output)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If COMPZ = 'N' or N < = 1 then LIWORK must be at least 1. If COMPZ = 'V' and N > 1 then LIWORK must be at least (6 + 6*N + 5*N*lg N). If COMPZ = 'I' and N > 1 then LIWORK must be at least (3 + 5*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an eigenvalue while
working on the submatrix lying in rows and columns
INFO/(N+1) through mod(INFO,N+1).
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

Modified by Francoise Tisseur, University of Tennessee.

NAME

sstegr - (a) Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

---

SYNOPSIS

```
 SUBROUTINE SSTEGR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*      W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEGR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*      M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEGR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: D, E, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEGR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL :: VL, VU, ABSTOL
REAL, DIMENSION(:) :: D, E, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstegr(char jobz, char range, int n, float *d, float *e, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *isuppz, int *info);

void sstegr_64(char jobz, char range, long n, float *d, float *e, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *isuppz, long *info);

PURPOSE

sstegr b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high relative accuracy by the dqds algorithm,

```
(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

Note 1 : Currently SSTEGR is only set up to find ALL the n eigenvalues and eigenvectors of T in `O(n^2)` time

Note 2 : Currently the routine SSTEIN is called when an appropriate sigma_i cannot be chosen in step (c) above. SSTEIN invokes modified Gram-Schmidt when eigenvalues are close.

Note 3 : SSTEGR works only on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs. Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the ieee standard.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

```
 = 'A': all eigenvalues will be found.

 = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix T. On exit, D is overwritten.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix T in elements 1 to N-1 of E; `E(N)` need not be set. On exit, E is overwritten.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues/eigenvectors. IF JOBZ = 'V', the eigenvalues and eigenvectors output have residual norms bounded by ABSTOL, and the dot products between different eigenvectors are bounded by ABSTOL. If ABSTOL is less than N*EPS*|T|, then N*EPS*|T| will be used in its place, where EPS is the machine precision and |T| is the 1-norm of the tridiagonal matrix. The eigenvalues are computed to an accuracy of EPS*|T| irrespective of ABSTOL. If high relative accuracy is important, set ABSTOL to DLAMCH('Safe minimum'). See Barlow and Demmel ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7 for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = `max(1,18*N)`

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**

On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= `max(1,10*N)`

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = 1, internal error in SLARRE,
if INFO  = 2, internal error in SLARRV.
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

sstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

---

SYNOPSIS

```
 SUBROUTINE SSTEIN( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 INTEGER N, M, LDZ, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEIN_64( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 INTEGER*8 N, M, LDZ, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEIN( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 INTEGER :: N, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL, DIMENSION(:) :: D, E, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEIN_64( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ],
*       [WORK], [IWORK], IFAIL, [INFO])
 INTEGER(8) :: N, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL, DIMENSION(:) :: D, E, W, WORK
 REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstein(int n, float *d, float *e, int m, float *w, int *iblock, int *isplit, float *z, int ldz, int *ifail, int *info);

void sstein_64(long n, float *d, float *e, long m, float *w, long *iblock, long *isplit, float *z, long ldz, long *ifail, long *info);

PURPOSE

sstein computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter MAXITS (currently set to 5).

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix T, in elements 1 to N-1. `E(N)` need not be set.
- **M (input)**
  The number of eigenvectors to be found. 0 < = M < = N.
- **W (input)**
  The first M elements of W contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array W from SSTEBZ with ORDER = 'B' is expected here.)
- **IBLOCK (input)**
  The submatrix indices associated with the corresponding eigenvalues in W; `IBLOCK(i)` =1 if eigenvalue `W(i)` belongs to the first submatrix from the top, =2 if `W(i)` belongs to the second submatrix, etc. (The output array IBLOCK from SSTEBZ is expected here.)
- **ISPLIT (input)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc. (The output array ISPLIT from SSTEBZ is expected here.)
- **Z (output)**
  The computed eigenvectors. The eigenvector associated with the eigenvalue `W(i)` is stored in the i-th column of Z. Any vector which fails to converge is set to its current iterate after MAXITS iterations.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK (workspace)**

```
dimension(N)
```

- **IFAIL (output)**
  On normal exit, all elements of IFAIL are zero. If one or more eigenvectors fail to converge after MAXITS iterations, then their indices are stored in array IFAIL.
- **INFO (output)**

```
 = 0: successful exit.

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, then i eigenvectors failed to converge
in MAXITS iterations.  Their indices are stored in
array IFAIL.
```

NAME

ssteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

---

SYNOPSIS

```
SUBROUTINE SSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER N, LDZ, INFO
REAL D(*), E(*), Z(LDZ,*), WORK(*)

SUBROUTINE SSTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
INTEGER*8 N, LDZ, INFO
REAL D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEQR( COMPZ, N, D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
REAL, DIMENSION(:,:) :: Z

SUBROUTINE STEQR_64( COMPZ, N, D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: D, E, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void ssteqr(char compz, int n, float *d, float *e, float *z, int ldz, int *info);

void ssteqr_64(char compz, long n, float *d, float *e, float *z, long ldz, long *info);

PURPOSE

ssteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. The eigenvectors of a full or band symmetric matrix can also be found if SSYTRD or SSPTRD or SSBTRD has been used to reduce this matrix to tridiagonal form.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvalues and eigenvectors of the original
symmetric matrix.  On entry, Z must contain the
orthogonal matrix used to reduce the original matrix
to tridiagonal form.
 = 'I':  Compute eigenvalues and eigenvectors of the
tridiagonal matrix.  Z is initialized to the identity
matrix.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the orthogonal matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original symmetric matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if eigenvectors are desired, then LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(max(1,2*N-2))` If COMPZ = 'N', then WORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm has failed to find all the eigenvalues in
a total of 30*N iterations; if INFO  = i, then i
elements of E have not converged to zero; on exit, D
and E contain the elements of a symmetric tridiagonal
matrix which is orthogonally similar to the original
matrix.
```

NAME

ssterf - compute all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm

---

SYNOPSIS

```
SUBROUTINE SSTERF( N, D, E, INFO)
INTEGER N, INFO
REAL D(*), E(*)

SUBROUTINE SSTERF_64( N, D, E, INFO)
INTEGER*8 N, INFO
REAL D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE STERF( [N], D, E, [INFO])
INTEGER :: N, INFO
REAL, DIMENSION(:) :: D, E

SUBROUTINE STERF_64( [N], D, E, [INFO])
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void ssterf(int n, float *d, float *e, int *info);

void ssterf_64(long n, float *d, float *e, long *info);

PURPOSE

ssterf computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm failed to find all of the eigenvalues in
a total of 30*N iterations; if INFO  = i, then i
elements of E have not converged to zero.
```

NAME

sstev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE SSTEV( JOBZ, N, DIAG, OFFD, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ
INTEGER N, LDZ, INFO
REAL DIAG(*), OFFD(*), Z(LDZ,*), WORK(*)

SUBROUTINE SSTEV_64( JOBZ, N, DIAG, OFFD, Z, LDZ, WORK, INFO)
CHARACTER * 1 JOBZ
INTEGER*8 N, LDZ, INFO
REAL DIAG(*), OFFD(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEV( JOBZ, [N], DIAG, OFFD, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ
INTEGER :: N, LDZ, INFO
REAL, DIMENSION(:) :: DIAG, OFFD, WORK
REAL, DIMENSION(:,:) :: Z

SUBROUTINE STEV_64( JOBZ, [N], DIAG, OFFD, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: JOBZ
INTEGER(8) :: N, LDZ, INFO
REAL, DIMENSION(:) :: DIAG, OFFD, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstev(char jobz, int n, float *diag, float *offd, float *z, int ldz, int *info);

void sstev_64(char jobz, long n, float *diag, float *offd, float *z, long ldz, long *info);

PURPOSE

sstev computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, if INFO = 0, the eigenvalues in ascending order.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A, stored in elements 1 to N-1 of OFFD; `OFFD(N)` need not be set, but is used by the routine. On exit, the contents of OFFD are destroyed.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with DIAG(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  If JOBZ = 'N', WORK is not referenced.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of OFFD did not converge to zero.
```

NAME

sstevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix

---

SYNOPSIS

```
 SUBROUTINE SSTEVD( JOBZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER N, LDZ, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL D(*), E(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEVD_64( JOBZ, N, D, E, Z, LDZ, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ
 INTEGER*8 N, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL D(*), E(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVD( JOBZ, [N], D, E, Z, [LDZ], [WORK], [LWORK], [IWORK],
*       [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEVD_64( JOBZ, [N], D, E, Z, [LDZ], [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 INTEGER(8) :: N, LDZ, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: D, E, WORK
 REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstevd(char jobz, int n, float *d, float *e, float *z, int ldz, int *info);

void sstevd_64(char jobz, long n, float *d, float *e, float *z, long ldz, long *info);

PURPOSE

sstevd computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A, stored in elements 1 to N-1 of E; `E(N)` need not be set, but is used by the routine. On exit, the contents of E are destroyed.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with D(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOBZ = 'N' or N < = 1 then LWORK must be at least 1. If JOBZ = 'V' and N > 1 then LWORK must be at least (1 + 4*N + N**2).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of the array IWORK. If JOBZ = 'N' or N < = 1 then LIWORK must be at least 1. If JOBZ = 'V' and N > 1 then LIWORK must be at least 3+5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of E did not converge to zero.
```

NAME

sstevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE SSTEVR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*      W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEVR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*      M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL D(*), E(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: D, E, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEVR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL :: VL, VU, ABSTOL
REAL, DIMENSION(:) :: D, E, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstevr(char jobz, char range, int n, float *d, float *e, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *isuppz, int *info);

void sstevr_64(char jobz, char range, long n, float *d, float *e, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *isuppz, long *info);

PURPOSE

sstevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, SSTEVR calls SSTEGR to compute the

eigenspectrum using Relatively Robust Representations. SSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
(a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
     is a relatively robust representation,

(b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
    relative accuracy by the dqds algorithm,

(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : SSTEVR calls SSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. SSTEVR calls SSTEBZ and SSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, D may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A in elements 1 to N-1 of E; E(N) need not be set. On exit, E may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

  If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but future releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**

The first M elements contain the selected eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 20*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal (and minimal) LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 10*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  Internal error
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA
```

NAME

sstevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A

SYNOPSIS

```
 SUBROUTINE SSTEVX( JOBZ, RANGE, N, DIAG, OFFD, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL DIAG(*), OFFD(*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSTEVX_64( JOBZ, RANGE, N, DIAG, OFFD, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL DIAG(*), OFFD(*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEVX( JOBZ, RANGE, [N], DIAG, OFFD, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: DIAG, OFFD, W, WORK
 REAL, DIMENSION(:,:) :: Z

 SUBROUTINE STEVX_64( JOBZ, RANGE, [N], DIAG, OFFD, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [IWORK2], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
```

```
REAL, DIMENSION(:) :: DIAG, OFFD, W, WORK
REAL, DIMENSION(:,:) :: Z
```

C INTERFACE

#include <sunperf.h>

void sstevx(char jobz, char range, int n, float *diag, float *offd, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, float *z, int ldz, int *ifail, int *info);

void sstevx_64(char jobz, char range, long n, float *diag, float *offd, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

sstevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, DIAG may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A in elements 1 to N-1 of OFFD; `OFFD(N)` need not be set. On exit, OFFD may be multiplied by a constant factor chosen to avoid over/underflow in computing the eigenvalues.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.

- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge (INFO > 0), then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

sstsv - compute the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix

---

SYNOPSIS

```
SUBROUTINE SSTSV( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)
REAL L(*), D(*), SUBL(*), B(LDB,*)

SUBROUTINE SSTSV_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
REAL L(*), D(*), SUBL(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE STSV( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL
REAL, DIMENSION(:,:) :: B

SUBROUTINE STSV_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void sstsv(int n, int nrhs, float *l, float *d, float *subl, float *b, int ldb, int *ipiv, int *info);

void sstsv_64(long n, long nrhs, float *l, float *d, float *subl, float *b, long ldb, long *ipiv, long *info);

PURPOSE

sstsv computes the solution to a system of linear equations A * X = B where A is a symmetric tridiagonal matrix.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides in B.
- **L (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.
- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.
- **SUBL (output)**

  ```
  REAL array, dimension (N)
  ```

  On exit, part of the factorization of A.
- **B (input/output)**
  The columns of B contain the right hand sides.
- **LDB (input)**
  The leading dimension of B as specified in a type or DIMENSION statement.
- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.
- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

 > 0: if INFO = i, D(k,k) is exactly zero. The factorization has
been completed, but the block diagonal matrix D is exactly singular
and division by zero will occur if it is used to solve a system of
equations.

NAME

ssttrf - compute the factorization of a symmetric tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE SSTTRF( N, L, D, SUBL, IPIV, INFO)
INTEGER N, INFO
INTEGER IPIV(*)
REAL L(*), D(*), SUBL(*)

SUBROUTINE SSTTRF_64( N, L, D, SUBL, IPIV, INFO)
INTEGER*8 N, INFO
INTEGER*8 IPIV(*)
REAL L(*), D(*), SUBL(*)
```

F95 INTERFACE

```
SUBROUTINE STTRF( [N], L, D, SUBL, IPIV, [INFO])
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL

SUBROUTINE STTRF_64( [N], L, D, SUBL, IPIV, [INFO])
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL
```

C INTERFACE

#include <sunperf.h>

void ssttrf(int n, float *l, float *d, float *subl, int *ipiv, int *info);

void ssttrf_64(long n, float *l, float *d, float *subl, long *ipiv, long *info);

PURPOSE

ssttrf computes the factorization of a complex Hermitian tridiagonal matrix A.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.

- **L (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.

- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the L*D*L**H factorization of A.

- **SUBL (output)**

  ```
  REAL array, dimension (N)
  ```

  On exit, part of the factorization of A.

- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.

- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, D(k,k) is exactly zero.  The factorization has
  been completed, but the block diagonal matrix D is exactly singular
  and division by zero will occur if it is used to solve a system of
  equations.
  ```

NAME

ssttrs - computes the solution to a real system of linear equations A * X = B

SYNOPSIS

```
SUBROUTINE SSTTRS( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)
REAL L(*), D(*), SUBL(*), B(LDB,*)

SUBROUTINE SSTTRS_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
REAL L(*), D(*), SUBL(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE STTRS( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL
REAL, DIMENSION(:,:) :: B

SUBROUTINE STTRS_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:) :: L, D, SUBL
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void ssttrs(int n, int nrhs, float *l, float *d, float *subl, float *b, int ldb, int *ipiv, int *info);

void ssttrs_64(long n, long nrhs, float *l, float *d, float *subl, float *b, long ldb, long *ipiv, long *info);

PURPOSE

ssttrs computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric tridiagonal matrix and X and B are N-by-NRHS matrices.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.

- **NRHS (input)**

  ```
  INTEGER
  ```

  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.

- **L (input)**

  ```
  REAL array, dimension (N-1)
  ```

  On entry, the subdiagonal elements of LL and DD.

- **D (input)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the diagonal elements of DD.

- **SUBL (input)**

  ```
  REAL array, dimension (N-2)
  ```

  On entry, the second subdiagonal elements of LL.

- **B (input)**

  ```
  REAL array, dimension
  ```

  (LDB, NRHS) On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.

- **LDB (input)**

  ```
  INTEGER
  ```

  The leading dimension of the array B. LDB > = max(1, N)

- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

Details of the interchanges and block pivot. If IPIV(K) > 0, 1 by 1 pivot, and if IPIV(K) = K + 1 an interchange done; If IPIV(K) < 0, 2 by 2 pivot, no interchange required.

- **INFO (output)**

```
INTEGER

= 0:  successful exit

< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

sswap - Exchange vectors x and y.

SYNOPSIS

```
SUBROUTINE SSWAP( N, X, INCX, Y, INCY)
INTEGER N, INCX, INCY
REAL X(*), Y(*)

SUBROUTINE SSWAP_64( N, X, INCX, Y, INCY)
INTEGER*8 N, INCX, INCY
REAL X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SWAP( [N], X, [INCX], Y, [INCY])
INTEGER :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y

SUBROUTINE SWAP_64( [N], X, [INCX], Y, [INCY])
INTEGER(8) :: N, INCX, INCY
REAL, DIMENSION(:) :: X, Y
```

C INTERFACE

#include <sunperf.h>

void sswap(int n, float *x, int incx, float *y, int incy);

void sswap_64(long n, float *x, long incx, float *y, long incy);

PURPOSE

sswap Exchange x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input/output)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, the y vector.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, the x vector.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

ssycon - estimate the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
 SUBROUTINE SSYCON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE SSYCON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      IWORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*), IWORK2(*)
 REAL ANORM, RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYCON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, IWORK2
 REAL :: ANORM, RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE SYCON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [IWORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, IWORK2
```

```
REAL :: ANORM, RCOND
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssycon(char uplo, int n, float *a, int lda, int *ipivot, float anorm, float *rcond, int *info);

void ssycon_64(char uplo, long n, float *a, long lda, long *ipivot, float anorm, float *rcond, long *info);

PURPOSE

ssycon estimates the reciprocal of the condition number (in the 1-norm) of a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **IWORK2 (workspace)**
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ssyev - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
SUBROUTINE SSYEV( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER N, LDA, LDWORK, INFO
REAL A(LDA,*), W(*), WORK(*)

SUBROUTINE SSYEV_64( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, INFO)
CHARACTER * 1 JOBZ, UPLO
INTEGER*8 N, LDA, LDWORK, INFO
REAL A(LDA,*), W(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYEV( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER :: N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: W, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYEV_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
INTEGER(8) :: N, LDA, LDWORK, INFO
REAL, DIMENSION(:) :: W, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssyev(char jobz, char uplo, int n, float *a, int lda, float *w, int *info);

void ssyev_64(char jobz, char uplo, long n, float *a, long lda, float *w, long *info);

PURPOSE

ssyev computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,3*N-1). For optimal efficiency, LDWORK >= (NB+2)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

ssyevd - compute all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
 SUBROUTINE SSYEVD( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER N, LDA, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL A(LDA,*), W(*), WORK(*)

 SUBROUTINE SSYEVD_64( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 N, LDA, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL A(LDA,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVD( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: N, LDA, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE SYEVD_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: N, LDA, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssyevd(char jobz, char uplo, int n, float *a, int lda, float *w, int *info);

void ssyevd_64(char jobz, char uplo, long n, float *a, long lda, float *w, long *info);

PURPOSE

ssyevd computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Because of large use of BLAS of level 3, SSYEVD needs N**2 more workspace than SSYEVX.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  dimension (LWORK) On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least 2*N+1. If JOBZ = 'V' and N > 1, LWORK must be at least 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK must be at least 1. If JOBZ = 'N' and N > 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

Modified by Francoise Tisseur, University of Tennessee.

NAME

ssyevr - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE SSYEVR( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSYEVR_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 REAL VL, VU, ABSTOL
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVR( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, Z

 SUBROUTINE SYEVR_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK],
*       [LIWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
INTEGER(8) :: N, LDA, IL, IU, M, LDZ, LWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL :: VL, VU, ABSTOL
REAL, DIMENSION(:) :: W, WORK
REAL, DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void ssyevr(char jobz, char range, char uplo, int n, float *a, int lda, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *isuppz, int *info);

void ssyevr_64(char jobz, char range, char uplo, long n, float *a, long lda, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *isuppz, long *info);

PURPOSE

ssyevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, SSYEVR calls SSTEGR to compute the

eigenspectrum using Relatively Robust Representations. SSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
(a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
     is a relatively robust representation,

(b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
    relative accuracy by the dqds algorithm,

(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : SSYEVR calls SSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. SSYEVR calls SSTEBZ and SSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of SSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.

- **VU (input)**
  See the description of VL.

- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.

- **IU (input)**
  See the description of IL.

- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but furutre releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,26*N). For optimal efficiency, LWORK > = (NB+6)*N, where NB is the max of the blocksize for SSYTRD and SORMTR returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK > = max(1,10*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  Internal error
```

FURTHER DETAILS

Based on contributions by

Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA

NAME

ssyevx - compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A

SYNOPSIS

```
 SUBROUTINE SSYEVX( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSYEVX_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, IWORK2, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER*8 IWORK2(*), IFAIL(*)
 REAL VL, VU, ABTOL
 REAL A(LDA,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYEVX( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, [LDZ], [WORK], [LDWORK], [IWORK2], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK2, IFAIL
 REAL :: VL, VU, ABTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, Z

 SUBROUTINE SYEVX_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABTOL, NFOUND, W, Z, [LDZ], [WORK], [LDWORK], [IWORK2],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK2, IFAIL
REAL :: VL, VU, ABTOL
REAL, DIMENSION(:) :: W, WORK
REAL, DIMENSION(:,:) :: A, Z
```

C INTERFACE

#include <sunperf.h>

void ssyevx(char jobz, char range, char uplo, int n, float *a, int lda, float vl, float vu, int il, int iu, float abtol, int *nfound, float *w, float *z, int ldz, int *ifail, int *info);

void ssyevx_64(char jobz, char range, char uplo, long n, float *a, long lda, float vl, float vu, long il, long iu, float abtol, long *nfound, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

ssyevx computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (output)**
  The total number of eigenvalues found. 0 <= NFOUND <= N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  On normal exit, the first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,8*N). For optimal efficiency, LDWORK >= (NB+3)*N, where NB is the max of the blocksize for SSYTRD and SORMTR returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **IWORK2 (workspace)**
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: if INFO = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.

NAME

ssygs2 - reduce a real symmetric-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE SSYGS2( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SSYGS2_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYGS2( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE SYGS2_64( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssygs2(int itype, char uplo, int n, float *a, int lda, float *b, int ldb, int *info);

void ssygs2_64(long itype, char uplo, long n, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

ssygs2 reduces a real symmetric-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U')*A*inv(U)` or `inv(L)*A*inv(L')`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U` or L'*A*L.

B must have been previously factorized as U'*U or L*L' by SPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U')*A*inv(U) or inv(L)*A*inv(L');

  = 2 or 3: compute U*A*U' or L'*A*L.
  ```

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored, and how B has been factorized. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by SPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

ssygst - reduce a real symmetric-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE SSYGST( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER ITYPE, N, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SSYGST_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 ITYPE, N, LDA, LDB, INFO
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYGST( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: ITYPE, N, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE SYGST_64( ITYPE, UPLO, [N], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssygst(int itype, char uplo, int n, float *a, int lda, float *b, int ldb, int *info);

void ssygst_64(long itype, char uplo, long n, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

ssygst reduces a real symmetric-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**T)*A*inv(U)` or `inv(L)*A*inv(L**T)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**T or L**T*A*L.

B must have been previously factorized as U**T*U or L*L**T by SPOTRF.

ARGUMENTS

- **ITYPE (input)**

```
= 1: compute inv(U**T)*A*inv(U) or inv(L)*A*inv(L**T);

= 2 or 3: compute U*A*U**T or L**T*A*L.
```

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored and B is factored as
U**T*U;
 = 'L':  Lower triangle of A is stored and B is factored as
L*L**T.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by SPOTRF.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ssygv - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSYGV( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), W(*), WORK(*)

 SUBROUTINE SSYGV_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL A(LDA,*), B(LDB,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGV( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE SYGV_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssygv(int itype, char jobz, char uplo, int n, float *a, int lda, float *b, int ldb, float *w, int *info);

void ssygv_64(long itype, char jobz, char uplo, long n, float *a, long lda, float *b, long ldb, float *w, long *info);

PURPOSE

ssygv computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also

positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **B (input/output)**
  On entry, the symmetric positive definite matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO <= N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,3*N-1). For optimal efficiency, LDWORK >= (NB+2)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEV returned an error code:

 <= N:  if INFO  = i, SSYEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

ssygvd - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSYGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL A(LDA,*), B(LDB,*), W(*), WORK(*)

 SUBROUTINE SSYGVD_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL A(LDA,*), B(LDB,*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGVD( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE SYGVD_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: W, WORK
```

```
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssygvd(int itype, char jobz, char uplo, int n, float *a, int lda, float *b, int ldb, float *w, int *info);

void ssygvd_64(long itype, char jobz, char uplo, long n, float *a, long lda, float *b, long ldb, float *w, long *info);

PURPOSE

ssygvd computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the symmetric matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N < = 1, LWORK > = 1. If JOBZ = 'N' and N > 1, LWORK > = 2*N+1. If JOBZ = 'V' and N > 1, LWORK > = 1 + 6*N + 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK > = 1. If JOBZ = 'N' and N > 1, LIWORK > = 1. If JOBZ = 'V' and N > 1, LIWORK > = 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEVD returned an error code:

 < = N:  if INFO  = i, SSYEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

ssygvx - compute selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE SSYGVX( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB, VL,
*      VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL A(LDA,*), B(LDB,*), W(*), Z(LDZ,*), WORK(*)

 SUBROUTINE SSYGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB,
*      VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 INTEGER*8 ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 REAL VL, VU, ABSTOL
 REAL A(LDA,*), B(LDB,*), W(*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYGVX( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B, [LDB],
*       VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK], [IWORK],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, B, Z
```

```
 SUBROUTINE SYGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B,
*       [LDB], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 INTEGER(8) :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL :: VL, VU, ABSTOL
 REAL, DIMENSION(:) :: W, WORK
 REAL, DIMENSION(:,:) :: A, B, Z
```

C INTERFACE

#include <sunperf.h>

void ssygvx(int itype, char jobz, char range, char uplo, int n, float *a, int lda, float *b, int ldb, float vl, float vu, int il, int iu, float abstol, int *m, float *w, float *z, int ldz, int *ifail, int *info);

void ssygvx_64(long itype, char jobz, char range, char uplo, long n, float *a, long lda, float *b, long ldb, float vl, float vu, long il, long iu, float abstol, long *m, float *w, float *z, long ldz, long *ifail, long *info);

PURPOSE

ssygvx computes selected eigenvalues, and optionally, eigenvectors of a real generalized symmetric-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
  ```

```
will be found.
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
 = 'U':  Upper triangle of A and B are stored;

 = 'L':  Lower triangle of A and B are stored.
```

- **N (input)**
  The order of the matrix pencil (A,B). N > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the symmetric matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**T*U or B = L*L**T.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  See the description of VL.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  See the description of IL.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*DLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the

orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,8*N). For optimal efficiency, LWORK > = (NB+3)*N, where NB is the blocksize for SSYTRD returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  SPOTRF or SSYEVX returned an error code:

 < = N:  if INFO  = i, SSYEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

ssymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE SSYMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 INTEGER M, N, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE SSYMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 INTEGER*8 M, N, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 INTEGER :: M, N, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C

 SUBROUTINE SYMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 INTEGER(8) :: M, N, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void ssymm(char side, char uplo, int m, int n, float alpha, float *a, int lda, float *b, int ldb, float beta, float *c, int ldc);

void ssymm_64(char side, char uplo, long m, long n, float alpha, float *a, long lda, float *b, long ldb, float beta, float *c, long ldc);

PURPOSE

ssymm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the symmetric matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the symmetric matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the symmetric matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the symmetric matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the symmetric matrix, such that when UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of

the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

ssymv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE SSYMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER N, LDA, INCX, INCY
REAL ALPHA, BETA
REAL A(LDA,*), X(*), Y(*)

SUBROUTINE SSYMV_64( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INCX, INCY
REAL ALPHA, BETA
REAL A(LDA,*), X(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE SYMV( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INCX, INCY
REAL :: ALPHA, BETA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYMV_64( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INCX, INCY
REAL :: ALPHA, BETA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssymv(char uplo, int n, float alpha, float *a, int lda, float *x, int incx, float beta, float *y, int incy);

void ssymv_64(char uplo, long n, float alpha, float *a, long lda, float *x, long incx, float beta, float *y, long incy);

PURPOSE

ssymv performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ssyr - perform the symmetric rank 1 operation A := alpha*x*x' + A

SYNOPSIS

```
SUBROUTINE SSYR( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
INTEGER N, INCX, LDA
REAL ALPHA
REAL X(*), A(LDA,*)

SUBROUTINE SSYR_64( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, LDA
REAL ALPHA
REAL X(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYR( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYR_64( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssyr(char uplo, int n, float alpha, float *x, int incx, float *a, int lda);

void ssyr_64(char uplo, long n, float alpha, float *x, long incx, float *a, long lda);

PURPOSE

ssyr performs the symmetric rank 1 operation A := alpha*x*x' + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

ssyr2 - perform the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A

SYNOPSIS

```
SUBROUTINE SSYR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
INTEGER N, INCX, INCY, LDA
REAL ALPHA
REAL X(*), Y(*), A(LDA,*)

SUBROUTINE SSYR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
INTEGER*8 N, INCX, INCY, LDA
REAL ALPHA
REAL X(*), Y(*), A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, INCX, INCY, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, INCX, INCY, LDA
REAL :: ALPHA
REAL, DIMENSION(:) :: X, Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssyr2(char uplo, int n, float alpha, float *x, int incx, float *y, int incy, float *a, int lda);

void ssyr2_64(char uplo, long n, float alpha, float *x, long incx, float *y, long incy, float *a, long lda);

PURPOSE

ssyr2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

ssyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

SYNOPSIS

```
 SUBROUTINE SSYR2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER N, K, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE SSYR2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER*8 N, K, LDA, LDB, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYR2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER :: N, K, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C

 SUBROUTINE SYR2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER(8) :: N, K, LDA, LDB, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void ssyr2k(char uplo, char transa, int n, int k, float alpha, float *a, int lda, float *b, int ldb, float beta, float *c, int ldc);

void ssyr2k_64(char uplo, char transa, long n, long k, float alpha, float *a, long lda, float *b, long ldb, float beta, float *c, long ldc);

PURPOSE

ssyr2k K performs one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*B' + alpha*B*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*B + alpha*B'*A + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*A'*B + alpha*B'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'T' or 't' or 'C' or 'c', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or

'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

ssyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE SSYRFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SSYRFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYRFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE SYRFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void ssyrfs(char uplo, int n, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void ssyrfs_64(char uplo, long n, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

ssyrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by SSYTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ssyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

SYNOPSIS

```
 SUBROUTINE SSYRK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER N, K, LDA, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), C(LDC,*)

 SUBROUTINE SSYRK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 INTEGER*8 N, K, LDA, LDC
 REAL ALPHA, BETA
 REAL A(LDA,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE SYRK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER :: N, K, LDA, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, C

 SUBROUTINE SYRK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 INTEGER(8) :: N, K, LDA, LDC
 REAL :: ALPHA, BETA
 REAL, DIMENSION(:,:) :: A, C
```

C INTERFACE

#include <sunperf.h>

void ssyrk(char uplo, char transa, int n, int k, float alpha, float *a, int lda, float beta, float *c, int ldc);

void ssyrk_64(char uplo, char transa, long n, long k, float alpha, float *a, long lda, float beta, float *c, long ldc);

PURPOSE

ssyrk performs one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*A + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*A'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'T' or 't' or 'C' or 'c', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**

On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

ssysv - compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SSYSV( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER IPIVOT(*)
 REAL A(LDA,*), B(LDB,*), WORK(*)

 SUBROUTINE SSYSV_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 INTEGER*8 N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
 REAL A(LDA,*), B(LDB,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSV( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE SYSV_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 INTEGER(8) :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssysv(char uplo, int n, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void ssysv_64(char uplo, long n, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

ssysv computes the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by SSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and

D(k-1:k,k-1:k) is a 2-by-2 diagonal block. If UPLO = 'L' and IPIVOT(k) = IPIVOT(k+1) < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and D(k:k+1,k:k+1) is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 1, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for SSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

ssysvx - use the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE SSYSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE SSYSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*), WORK2(*)
 REAL RCOND
 REAL A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT, WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, AF, B, X

 SUBROUTINE SYSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IPIVOT, WORK2
REAL :: RCOND
REAL, DIMENSION(:) :: FERR, BERR, WORK
REAL, DIMENSION(:,:) :: A, AF, B, X
```

C INTERFACE

#include <sunperf.h>

void ssysvx(char fact, char uplo, int n, int nrhs, float *a, int lda, float *af, int ldaf, int *ipivot, float *b, int ldb, float *x, int ldx, float *rcond, float *ferr, float *berr, int *info);

void ssysvx_64(char fact, char uplo, long n, long nrhs, float *a, long lda, float *af, long ldaf, long *ipivot, float *b, long ldb, float *x, long ldx, float *rcond, float *ferr, float *berr, long *info);

PURPOSE

ssysvx uses the diagonal pivoting factorization to compute the solution to a real system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',

where U (or L) is a product of permutation and unit upper (lower)
triangular matrices, and D is symmetric and block diagonal with
1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by SSYTRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by SSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by SSYTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**

The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).

- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK >= 3*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for SSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **WORK2 (workspace)**
  `dimension(N)`

- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

ssytd2 - reduce a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE SSYTD2( UPLO, N, A, LDA, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
REAL A(LDA,*), D(*), E(*), TAU(*)

SUBROUTINE SSYTD2_64( UPLO, N, A, LDA, D, E, TAU, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
REAL A(LDA,*), D(*), E(*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE SYTD2( UPLO, [N], A, [LDA], D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:) :: D, E, TAU
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYTD2_64( UPLO, [N], A, [LDA], D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:) :: D, E, TAU
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssytd2(char uplo, int n, float *a, int lda, float *d, float *e, float *tau, int *info);

void ssytd2_64(char uplo, long n, float *a, long lda, float *d, float *e, float *tau, long *info);

PURPOSE

ssytd2 reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: Q' * A * Q = T.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in

A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

ssytf2 - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE SSYTF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
INTEGER IPIV(*)
REAL A(LDA,*)

SUBROUTINE SSYTF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE SYTF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYTF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssytf2(char uplo, int n, float *a, int lda, int *ipiv, int *info);

void ssytf2_64(char uplo, long n, float *a, long lda, long *ipiv, long *info);

PURPOSE

ssytf2 computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the transpose of U, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

ssytrd - reduce a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation

---

SYNOPSIS

```
SUBROUTINE SSYTRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LWORK, INFO
REAL A(LDA,*), D(*), E(*), TAU(*), WORK(*)

SUBROUTINE SSYTRD_64( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LWORK, INFO
REAL A(LDA,*), D(*), E(*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRD( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: D, E, TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYTRD_64( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: D, E, TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssytrd(char uplo, int n, float *a, int lda, float *d, float *e, float *tau, int *info);

void ssytrd_64(char uplo, long n, float *a, long lda, float *d, float *e, float *tau, long *info);

PURPOSE

ssytrd reduces a real symmetric matrix A to real symmetric tridiagonal form T by an orthogonal similarity transformation: Q**T * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. For optimum performance LWORK >= N*NB, where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in

A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real scalar, and v is a real vector with

`v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

ssytrf - compute the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE SSYTRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), WORK(*)

SUBROUTINE SSYTRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYTRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssytrf(char uplo, int n, float *a, int lda, int *ipivot, int *info);

void ssytrf_64(char uplo, long n, float *a, long lda, long *ipivot, long *info);

PURPOSE

ssytrf computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    v    0   )   k-s

U(k)  =  (   0    I    0   )   s

         (   0    0    I   )   n-k

            k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    0     0   )  k-1

L(k)  =  (   0    I     0   )  s
```

```
(   0    v     I   )  n-k-s+1

   k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

ssytri - compute the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
SUBROUTINE SSYTRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), WORK(*)

SUBROUTINE SSYTRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE SYTRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void ssytri(char uplo, int n, float *a, int lda, int *ipivot, int *info);

void ssytri_64(char uplo, long n, float *a, long lda, long *ipivot, long *info);

PURPOSE

ssytri computes the inverse of a real symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

ssytrs - solve a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF

SYNOPSIS

```
SUBROUTINE SSYTRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)
REAL A(LDA,*), B(LDB,*)

SUBROUTINE SSYTRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE SYTRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B

SUBROUTINE SYTRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void ssytrs(char uplo, int n, int nrhs, float *a, int lda, int *ipivot, float *b, int ldb, int *info);

void ssytrs_64(char uplo, long n, long nrhs, float *a, long lda, long *ipivot, float *b, long ldb, long *info);

PURPOSE

ssytrs solves a system of linear equations A*X = B with a real symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by SSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by SSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by SSYTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

stbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE STBCON( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER N, NDIAG, LDA, INFO
 INTEGER WORK2(*)
 REAL RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE STBCON_64( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND,
*      WORK, WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER*8 N, NDIAG, LDA, INFO
 INTEGER*8 WORK2(*)
 REAL RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TBCON( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER :: N, NDIAG, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE TBCON_64( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER(8) :: N, NDIAG, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
```

```
REAL :: RCOND
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void stbcon(char norm, char uplo, char diag, int n, int ndiag, float *a, int lda, float *rcond, int *info);

void stbcon_64(char norm, char uplo, char diag, long n, long ndiag, float *a, long lda, float *rcond, long *info);

PURPOSE

stbcon estimates the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

stbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE STBMV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NDIAG, LDA, INCY
REAL A(LDA,*), Y(*)

SUBROUTINE STBMV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NDIAG, LDA, INCY
REAL A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TBMV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NDIAG, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE TBMV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NDIAG, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void stbmv(char uplo, char transa, char diag, int n, int ndiag, float *a, int lda, float *y, int incy);

void stbmv_64(char uplo, char transa, char diag, long n, long ndiag, float *a, long lda, float *y, long incy);

PURPOSE

stbmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The

top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

stbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE STBRFS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE STBRFS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TBRFS( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, B, X

 SUBROUTINE TBRFS_64( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A,
*       [LDA], B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, B, X
```

C INTERFACE

#include <sunperf.h>

void stbrfs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, float *a, int lda, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void stbrfs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, float *a, long lda, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

stbrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix.

The solution matrix X must be computed by STBTRS or some other means before entering this routine. STBRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(3*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

stbsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE STBSV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NDIAG, LDA, INCY
REAL A(LDA,*), Y(*)

SUBROUTINE STBSV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NDIAG, LDA, INCY
REAL A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TBSV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NDIAG, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE TBSV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NDIAG, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void stbsv(char uplo, char transa, char diag, int n, int ndiag, float *a, int lda, float *y, int incy);

void stbsv_64(char uplo, char transa, char diag, long n, long ndiag, float *a, long lda, float *y, long incy);

PURPOSE

stbsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper

triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

stbtrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
 SUBROUTINE STBTRS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*       LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NDIAG, NRHS, LDA, LDB, INFO
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE STBTRS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*       LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TBTRS( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*        B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE TBTRS_64( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*        B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void stbtrs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void stbtrs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

stbtrs solves a triangular system of the form

where A is a triangular band matrix of order N, and B is an N-by NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of A. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd). If DIAG = 'U', the diagonal elements of A

are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.
```

NAME

stgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B)

SYNOPSIS

```
 SUBROUTINE STGEVC( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 REAL A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE STGEVC_64( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 REAL A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEVC( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE TGEVC_64( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void stgevc(char side, char howmny, logical *select, int n, float *a, int lda, float *b, int ldb, float *vl, int ldvl, float *vr, int ldvr, int mm, int *m, int *info);

void stgevc_64(char side, char howmny, logical *select, long n, float *a, long lda, float *b, long ldb, float *vl, long ldvl, float *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

stgevc computes some or all of the right and/or left generalized eigenvectors of a pair of real upper triangular matrices (A,B).

The right generalized eigenvector x and the left generalized eigenvector y of (A,B) corresponding to a generalized eigenvalue w are defined by:

```
(A - wB) * x = 0  and  y**H * (A - wB) = 0
```

where y**H denotes the conjugate tranpose of y.

If an eigenvalue w is determined by zero diagonal elements of both A and B, a unit vector is returned as the corresponding eigenvector.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of (A,B), or the products Z*X and/or Q*Y, where Z and Q are input orthogonal matrices. If (A,B) was obtained from the generalized real-Schur factorization of an original pair of matrices

```
(A0,B0) = (Q*A*Z**H,Q*B*Z**H),
```

then Z*X and Q*Y are the matrices of right or left eigenvectors of A.

A must be block upper triangular, with 1-by-1 and 2-by-2 diagonal blocks. Corresponding to each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one

eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R': compute right eigenvectors only;

  = 'L': compute left eigenvectors only;

  = 'B': compute both right and left eigenvectors.
  ```
- **HOWMNY (input)**

```
 = 'A': compute all right and/or left eigenvectors;

 = 'B': compute all right and/or left eigenvectors, and
backtransform them using the input matrices supplied
in VR and/or VL;
 = 'S': compute selected right and/or left eigenvectors,
specified by the logical array SELECT.
```

- **SELECT (input)**
  If HOWMNY ='S', SELECT specifies the eigenvectors to be computed. If HOWMNY ='A' or 'B', SELECT is not referenced. To select the real eigenvector corresponding to the real eigenvalue w(j), `SELECT(j)` must be set to .TRUE. To select the complex eigenvector corresponding to a complex conjugate pair `w(j)` and w(j+1), either `SELECT(j)` or `SELECT(j+1)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A.
- **LDA (input)**
  The leading dimension of array A. LDA >= max(1, N).
- **B (input)**
  The upper triangular matrix B. If A has a 2-by-2 diagonal block, then the corresponding 2-by-2 block of B must be diagonal with positive elements.
- **LDB (input)**
  The leading dimension of array B. LDB >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the orthogonal matrix Q of left Schur vectors returned by SHGEQZ). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of (A,B); if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.

  A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.
- **LDVL (input)**
  The leading dimension of array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the orthogonal matrix Z of right Schur vectors returned by SHGEQZ). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of (A,B); if HOWMNY = 'B', the matrix Z*X; if HOWMNY = 'S', the right eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.

  A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the 2-by-2 block (INFO:INFO+1) does not have a complex
eigenvalue.
```

FURTHER DETAILS

Allocation of workspace:

---------- -- ---------

```
WORK( j )  = 1-norm of j-th column of A, above the diagonal
WORK( N+j )  = 1-norm of j-th column of B, above the diagonal
WORK( 2*N+1:3*N )  = real part of eigenvector

WORK( 3*N+1:4*N )  = imaginary part of eigenvector

WORK( 4*N+1:5*N )  = real part of back-transformed eigenvector
WORK( 5*N+1:6*N )  = imaginary part of back-transformed eigenvector
```

Rowwise vs. columnwise solution methods:

------- -- ---------- -------- -------

Finding a generalized eigenvector consists basically of solving the singular triangular system

```
 (A - w B) x  = 0     (for right) or:   (A - w B)**H y  = 0  (for left)
```

Consider finding the i-th right eigenvector (assume all eigenvalues are real). The equation to be solved is: i

0 = sum `C(j,k)` `v(k)` = sum `C(j,k)` `v(k)` for j = i,. . .,1 k =j k =j

where C = (A - w B) (The components `v(i+1:n)` are 0.)

The ``rowwise'' method is:

(1) `v(i)` : = 1

for j = i-1,. . .,1:

```
                       i

    (2) compute  s  = - sum C(j,k) v(k)   and

                      k =j+1

    (3) v(j) : = s / C(j,j)
```

Step 2 is sometimes called the ``dot product'' step, since it is an inner product between the j-th row and the portion of the eigenvector that has been computed so far.

The ``columnwise'' method consists basically in doing the sums for all the rows in parallel. As each `v(j)` is computed, the contribution of `v(j)` times the j-th column of C is added to the partial sums. Since FORTRAN arrays are stored columnwise, this has the advantage that at each step, the elements of C that are accessed are adjacent to one another, whereas with the rowwise method, the elements accessed at a step are spaced LDA (and LDB) words apart.

When finding left eigenvectors, the matrix in question is the transpose of the one in storage, so the rowwise method then actually accesses columns of A and B at each step, and so is the preferred method.

NAME

stgexc - reorder the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation (A, B) = Q * (A, B) * Z',

---

SYNOPSIS

```
 SUBROUTINE STGEXC( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z, LDZ,
*       IFST, ILST, WORK, LWORK, INFO)
 INTEGER N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL WANTQ, WANTZ
 REAL A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)

 SUBROUTINE STGEXC_64( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z,
*       LDZ, IFST, ILST, WORK, LWORK, INFO)
 INTEGER*8 N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL*8 WANTQ, WANTZ
 REAL A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEXC( WANTQ, WANTZ, N, A, [LDA], B, [LDB], Q, [LDQ], Z,
*        [LDZ], IFST, ILST, [WORK], [LWORK], [INFO])
 INTEGER :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL :: WANTQ, WANTZ
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE TGEXC_64( WANTQ, WANTZ, N, A, [LDA], B, [LDB], Q, [LDQ],
*        Z, [LDZ], IFST, ILST, [WORK], [LWORK], [INFO])
 INTEGER(8) :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, LWORK, INFO
 LOGICAL(8) :: WANTQ, WANTZ
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void stgexc(logical wantq, logical wantz, int n, float *a, int lda, float *b, int ldb, float *q, int ldq, float *z, int ldz, int *ifst, int *ilst, int *info);

void stgexc_64(logical wantq, logical wantz, long n, float *a, long lda, float *b, long ldb, float *q, long ldq, float *z, long ldz, long *ifst, long *ilst, long *info);

PURPOSE

stgexc reorders the generalized real Schur decomposition of a real matrix pair (A,B) using an orthogonal equivalence transformation

so that the diagonal block of (A, B) with row index IFST is moved to row ILST.

(A, B) must be in generalized real Schur canonical form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

Optionally, the matrices Q and Z of generalized Schur vectors are updated.

```
Q(in) * A(in) * Z(in)' = Q(out) * A(out) * Z(out)'
Q(in) * B(in) * Z(in)' = Q(out) * B(out) * Z(out)'
```

ARGUMENTS

- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the matrix A in generalized real Schur canonical form. On exit, the updated matrix A, again in generalized real Schur canonical form.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in generalized real Schur canonical form (A,B). On exit, the updated matrix B, again in generalized real Schur canonical form (A,B).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **Q (input/output)**

On entry, if WANTQ = .TRUE., the orthogonal matrix Q. On exit, the updated matrix Q. If WANTQ = .FALSE., Q is not referenced.

- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If WANTQ = .TRUE., LDQ > = N.
- **Z (input/output)**
  On entry, if WANTZ = .TRUE., the orthogonal matrix Z. On exit, the updated matrix Z. If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If WANTZ = .TRUE., LDZ > = N.
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of (A, B). The block with row index IFST is moved to row ILST, by a sequence of swapping between adjacent blocks. On exit, if IFST pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; ILST always points to the first row of the block in its final position (which may differ from its input value by +1 or -1). 1 < = IFST, ILST < = N.
- **ILST (input/output)**
  See the description of IFST.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 4*N + 16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 =0:  successful exit.

 <0:  if INFO  = -i, the i-th argument had an illegal value.

 =1:  The transformed matrix pair (A, B) would be too far
from generalized Schur form; the problem is ill-
conditioned. (A, B) may have been partially reordered,
and ILST points to the first row of the current
position of the block being moved.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

NAME

stgsen - reorder the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence transformation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B

SYNOPSIS

```
 SUBROUTINE STGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 INTEGER IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL WANTQ, WANTZ
 LOGICAL SELECT(*)
 REAL PL, PR
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), DIF(*), WORK(*)

 SUBROUTINE STGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHAR, ALPHAI, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK,
*      LWORK, IWORK, LIWORK, INFO)
 INTEGER*8 IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 WANTQ, WANTZ
 LOGICAL*8 SELECT(*)
 REAL PL, PR
 REAL A(LDA,*), B(LDB,*), ALPHAR(*), ALPHAI(*), BETA(*), Q(LDQ,*), Z(LDZ,*), DIF(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B, [LDB],
*       ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
 INTEGER :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL :: WANTQ, WANTZ
 LOGICAL, DIMENSION(:) :: SELECT
 REAL :: PL, PR
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, DIF, WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z

 SUBROUTINE TGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B,
*       [LDB], ALPHAR, ALPHAI, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF,
```

```
*       [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 INTEGER(8) :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8) :: WANTQ, WANTZ
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL :: PL, PR
 REAL, DIMENSION(:) :: ALPHAR, ALPHAI, BETA, DIF, WORK
 REAL, DIMENSION(:,:) :: A, B, Q, Z
```

C INTERFACE

#include <sunperf.h>

void stgsen(int ijob, logical wantq, logical wantz, logical *select, int n, float *a, int lda, float *b, int ldb, float *alphar, float *alphai, float *beta, float *q, int ldq, float *z, int ldz, int *m, float *pl, float *pr, float *dif, int *info);

void stgsen_64(long ijob, logical wantq, logical wantz, logical *select, long n, float *a, long lda, float *b, long ldb, float *alphar, float *alphai, float *beta, float *q, long ldq, float *z, long ldz, long *m, float *pl, float *pr, float *dif, long *info);

PURPOSE

stgsen reorders the generalized real Schur decomposition of a real matrix pair (A, B) (in terms of an orthonormal equivalence transformation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix A and the upper triangular B. The leading columns of Q and Z form orthonormal bases of the corresponding left and right eigenspaces (deflating subspaces). (A, B) must be in generalized real Schur canonical form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

STGSEN also computes the generalized eigenvalues

```
w(j) = (ALPHAR(j) + i*ALPHAI(j))/BETA(j)
```

of the reordered matrix pair (A, B).

Optionally, STGSEN computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[(A11,B11), (A22,B22)] and Difl[(A11,B11), (A22,B22)], i.e. the `separation(s)` between the matrix pairs (A11, B11) and (A22,B22) that correspond to the selected cluster and the eigenvalues outside the cluster, resp., and norms of ``projections'' onto left and right eigenspaces w.r.t. the selected cluster in the (1,1)-block.

ARGUMENTS

- **IJOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (PL and PR) or the deflating subspaces (Difu and Difl):

  ```
   =0: Only reorder w.r.t. SELECT. No extras.
  ```

  ```
   =1: Reciprocal of norms of "projections" onto left and right
  eigenspaces w.r.t. the selected cluster (PL and PR).
   =2: Upper bounds on Difu and Difl. F-norm-based estimate
  ```

  (DIF(1:2)).

  ```
   =3: Estimate of Difu and Difl. 1-norm-based estimate
  ```

  (DIF(1:2)). About 5 times as expensive as IJOB = 2. =4: Compute PL, PR and DIF (i.e. 0, 1 and 2 above): Economic version to get it all. =5: Compute PL, PR and DIF (i.e. 0, 1 and 3 above)

- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select a complex conjugate pair of eigenvalues `w(j)` and w(j+1), corresponding to a 2-by-2 diagonal block, either `SELECT(j)` or `SELECT(j+1)` or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper quasi-triangular matrix A, with (A, B) in generalized real Schur canonical form. On exit, A is overwritten by the reordered matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the upper triangular matrix B, with (A, B) in generalized real Schur canonical form. On exit, B is overwritten by the reordered matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ALPHAR (output)**
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHAR(j)` + ALPHAI(j)*i and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A,B) were further reduced to triangular form using complex unitary transformations. If `ALPHAI(j)` is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.
- **ALPHAI (output)**
  See the description of ALPHAR.
- **BETA (output)**
  See the description of ALPHAR.
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., Q is an N-by-N matrix. On exit, Q has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B); The leading M columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; and if WANTQ = .TRUE., LDQ >= N.
- **Z (input/output)**
  On entry, if WANTZ = .TRUE., Z is an N-by-N matrix. On exit, Z has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B); The leading M columns of Z form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1; If WANTZ = .TRUE., LDZ >= N.
- **M (output)**
  The dimension of the specified pair of left and right eigen- spaces (deflating subspaces). 0 <= M <= N.
- **PL (output)**
  If IJOB = 1, 4 or 5, PL, PR are lower bounds on the reciprocal of the norm of ``projections'' onto left and right eigenspaces with respect to the selected cluster. 0 < PL, PR <= 1. If M = 0 or M = N, PL = PR = 1. If IJOB = 0, 2 or 3, PL and PR are not referenced.
- **PR (output)**
  See the description of PL.
- **DIF (output)**
  If IJOB >= 2, `DIF(1:2)` store the estimates of Difu and Difl.

  If IJOB = 2 or 4, `DIF(1:2)` are F-norm-based upper bounds on

Difu and Difl. If IJOB = 3 or 5, DIF(1:2) are 1-norm-based estimates of Difu and Difl. If M = 0 or N, DIF(1:2) = F-norm([A, B]). If IJOB = 0 or 1, DIF is not referenced.

- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 4*N+16. If IJOB = 1, 2 or 4, LWORK >= MAX(4*N+16, 2*M*(N-M)). If IJOB = 3 or 5, LWORK >= MAX(4*N+16, 4*M*(N-M)).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced. Otherwise, on exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 1. If IJOB = 1, 2 or 4, LIWORK >= N+6. If IJOB = 3 or 5, LIWORK >= MAX(2*M*(N-M), N+6).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 =0: Successful exit.

 <0: If INFO  = -i, the i-th argument had an illegal value.

 =1: Reordering of (A, B) failed because the transformed
matrix pair (A, B) would be too far from generalized
Schur form; the problem is very ill-conditioned.
(A, B) may have been partially reordered.
If requested, 0 is returned in DIF(*), PL and PR.
```

FURTHER DETAILS

STGSEN first collects the selected eigenvalues by computing orthogonal U and W that move them to the top left corner of (A, B). In other words, the selected eigenvalues are the eigenvalues of (A11, B11) in:

```
U'*(A, B)*W  = (A11 A12) (B11 B12) n1

               ( 0  A22),( 0  B22) n2

                 n1  n2    n1  n2
```

where N = n1+n2 and U' means the transpose of U. The first n1 columns of U and W span the specified pair of left and right eigenspaces (deflating subspaces) of (A, B).

If (A, B) has been obtained from the generalized real Schur decomposition of a matrix pair (C, D) = Q*(A, B)*Z', then the reordered generalized real Schur form of (C, D) is given by

```
(C, D)  = (Q*U)*(U'*(A, B)*W)*(Z*W)',
```

and the first n1 columns of Q*U and Z*W span the corresponding deflating subspaces of (C, D) (Q and Z store Q*U and Z*W, resp.).

Note that if the selected eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

The reciprocal condition numbers of the left and right eigenspaces spanned by the first n1 columns of U and W (or Q*U and Z*W) may be returned in DIF(1:2), corresponding to Difu and Difl, resp.

The Difu and Difl are defined as:

ifu[(A11, B11), (A22, B22)] = sigma-min(Zu)

and ifl[(A11, B11), (A22, B22)] = Difu[(A22, B22), (A11, B11)],

where sigma-min(Zu) is the smallest singular value of the (2*n1*n2)-by-(2*n1*n2) matrix

u = [kron(In2, A11) -kron(A22', In1)]

```
          [ kron(In2, B11)  -kron(B22', In1) ].
```

Here, Inx is the identity matrix of size nx and A22' is the transpose of A22. kron(X, Y) is the Kronecker product between the matrices X and Y.

When DIF(2) is small, small changes in (A, B) can cause large changes in the deflating subspace. An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is PS * norm((A, B)) / DIF(2),

where EPS is the machine precision.

The reciprocal norm of the projectors on the left and right eigenspaces associated with (A11, B11) may be returned in PL and PR. They are computed as follows. First we compute L and R so that P*(A, B)*Q is block diagonal, where

```
  = ( I -L ) n1           Q  = ( I R ) n1

         ( 0  I ) n2    and        ( 0 I ) n2

           n1 n2                     n1 n2
```

and (L, R) is the solution to the generalized Sylvester equation 11*R - L*A22 = -A12 11*R - L*B22 = -B12

Then PL = (F-norm(L)**2+1)**(-1/2) and PR = (F-norm(R)**2+1)**(-1/2). An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

PS * norm((A, B)) / PL.

There are also global error bounds which valid for perturbations up to a certain restriction: A lower bound (x) on the smallest F-norm(E,F) for which an eigenvalue of (A11, B11) may move and coalesce with an eigenvalue of (A22, B22) under perturbation (E,F), (i.e. (A + E, B + F), is

```
 x  = min(Difu,Difl)/((1/(PL*PL)+1/(PR*PR))**(1/2)+2*max(1/PL,1/PR)).
```

An approximate bound on x can be computed from DIF(1:2), PL and PR.

If y = (F-norm(E,F) / x) < = 1, the angles between the perturbed (L', R') and unperturbed (L, R) left and right deflating subspaces associated with the selected cluster in the (1,1)-blocks can be bounded as

```
max-angle(L, L')  < = arctan( y * PL / (1 - y * (1 - PL * PL)**(1/2))
max-angle(R, R')  < = arctan( y * PR / (1 - y * (1 - PR * PR)**(1/2))
```

See LAPACK User's Guide section 4.11 or the following references for more information.

Note that if the default method for computing the Frobenius-norm- based estimate DIF is not wanted (see SLATDF), then the parameter IDIFJB (see below) should be changed from 3 to 4 (routine SLATDF (IJOB = 2 will be used)). See STGSYL for more details.

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

References

```
 = = = = = = = = = =
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software,

```
Report UMINF - 94.04, Department of Computing Science, Umea
University, S-901 87 Umea, Sweden, 1994. Also as LAPACK Working
Note 87. To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

stgsja - compute the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B

SYNOPSIS

```
 SUBROUTINE STGSJA( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B, LDB,
*      TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, NCYCLE,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL TOLA, TOLB
 REAL A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)

 SUBROUTINE STGSJA_64( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B,
*      LDB, TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK,
*      NCYCLE, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 INTEGER*8 M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL TOLA, TOLB
 REAL A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSJA( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL :: TOLA, TOLB
 REAL, DIMENSION(:) :: ALPHA, BETA, WORK
 REAL, DIMENSION(:,:) :: A, B, U, V, Q

 SUBROUTINE TGSJA_64( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 INTEGER(8) :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
```

```
REAL :: TOLA, TOLB
REAL, DIMENSION(:) :: ALPHA, BETA, WORK
REAL, DIMENSION(:,:) :: A, B, U, V, Q
```

C INTERFACE

#include <sunperf.h>

void stgsja(char jobu, char jobv, char jobq, int m, int p, int n, int k, int l, float *a, int lda, float *b, int ldb, float tola, float tolb, float *alpha, float *beta, float *u, int ldu, float *v, int ldv, float *q, int ldq, int *ncycle, int *info);

void stgsja_64(char jobu, char jobv, char jobq, long m, long p, long n, long k, long l, float *a, long lda, float *b, long ldb, float tola, float tolb, float *alpha, float *beta, float *u, long ldu, float *v, long ldv, float *q, long ldq, long *ncycle, long *info);

PURPOSE

stgsja computes the generalized singular value decomposition (GSVD) of two real upper triangular (or trapezoidal) matrices A and B.

On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine SGGSVP from a general M-by-N matrix A and P-by-N matrix B:

```
             N-K-L  K    L

A =    K ( 0    A12  A13 ) if M-K-L >= 0;

       L ( 0     0   A23 )

   M-K-L ( 0     0    0  )

            N-K-L  K    L

A =  K ( 0    A12  A13 ) if M-K-L < 0;

   M-K ( 0     0   A23 )

            N-K-L  K    L

B =  L ( 0     0   B13 )

   P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal.

On exit,

```
            U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R ),
```

where U, V and Q are orthogonal matrices, Z' denotes the transpose of Z, R is a nonsingular upper triangular matrix, and D1

and D2 are ``diagonal'' matrices, which are of the following structures:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
            M-K-L ( 0  0 )

                  K  L
       D2 = L   ( 0  S )
            P-L ( 0  0 )

               N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 ) K
            L (  0    0   R22 ) L
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),
  S = diag( BETA(K+1),  ... , BETA(K+L) ),
  C**2 + S**2 = I.
  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                K M-K K+L-M
    D1 =   K ( I  0    0   )
         M-K ( 0  C    0   )

                  K M-K K+L-M
    D2 =   M-K ( 0  S    0   )
         K+L-M ( 0  0    I   )
           P-L ( 0  0    0   )

               N-K-L  K   M-K  K+L-M
```

```
          M-K ( 0     0   R22  R23  )
        K+L-M ( 0     0    0   R33  )
```

where

C = diag(ALPHA(K+1), ... , ALPHA(M)),

S = diag(BETA(K+1), ... , BETA(M)),

C**2 + S**2 = I.

R = (R11 R12 R13) is stored in A(1:M, N-K-L+1:N) and R33 is stored (0 R22 R23)

in B(M-K+1:L,N+M-K-L+1:N) on exit.

The computation of the orthogonal transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U1, V1, or Q1.

STGSJA essentially uses a variant of Kogbetliantz algorithm to reduce min(L,M-K)-by-L triangular (or trapezoidal) matrix A23 and L-by-L matrix B13 to the form:

```
U1'*A13*Q1 = C1*R1; V1'*B13*Q1 = S1*R1,
```

where U1, V1 and Q1 are orthogonal matrix, and Z' is the transpose of Z. C1 and S1 are diagonal matrices satisfying

```
C1**2 + S1**2 = I,
```

and R1 is an L-by-L nonsingular upper triangular matrix.

ARGUMENTS

- **JOBU (input)**

```
 = 'U':  U must contain an orthogonal matrix U1 on entry, and
the product U1*U is returned;
 = 'I':  U is initialized to the unit matrix, and the
orthogonal matrix U is returned;
 = 'N':  U is not computed.
```

- **JOBV (input)**

```
 = 'V':  V must contain an orthogonal matrix V1 on entry, and
the product V1*V is returned;
 = 'I':  V is initialized to the unit matrix, and the
orthogonal matrix V is returned;
 = 'N':  V is not computed.
```

- **JOBQ (input)**

```
 = 'Q':  Q must contain an orthogonal matrix Q1 on entry, and
the product Q1*Q is returned;
 = 'I':  Q is initialized to the unit matrix, and the
orthogonal matrix Q is returned;
```

= 'N': Q is not computed.

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **K (input)**
  K and L specify the subblocks in the input matrices A and B:

  A23 = A(K+1:MIN(K+L,M),N-L+1:N) and B13 = B(1:L,N-L+1:N) of A and B, whose GSVD is going to be computed by STGSJA. See Further details.
- **L (input)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A(N-K+1:N,1:MIN(K+L,M)) contains the triangular matrix R or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, if necessary, B(M-K+1:L,N+M-K-L+1:N) contains a part of R. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the convergence criteria for the Jacobi- Kogbetliantz iteration procedure. Generally, they are the same as used in the preprocessing step, say TOLA = max(M,N)*norm(A)*MACHEPS, TOLB = max(P,N)*norm(B)*MACHEPS.
- **TOLB (input)**
  See the description of TOLA.
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L >= 0, ALPHA(K+1:K+L) = diag(C),

  BETA(K+1:K+L) = diag(S), or if M-K-L < 0, ALPHA(K+1:M) = C, ALPHA(M+1:K+L) = 0

  BETA(K+1:M) = S, BETA(M+1:K+L) = 1. Furthermore, if K+L < N, ALPHA(K+L+1:N) = 0 and

  BETA(K+L+1:N) = 0.
- **BETA (output)**
  See the description of ALPHA.
- **U (input/output)**
  On entry, if JOBU = 'U', U must contain a matrix U1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBU = 'I', U contains the orthogonal matrix U; if JOBU = 'U', U contains the product U1*U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,M) if JOBU = 'U'; LDU >= 1 otherwise.
- **V (input/output)**
  On entry, if JOBV = 'V', V must contain a matrix V1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBV = 'I', V contains the orthogonal matrix V; if JOBV = 'V', V contains the product V1*V. If JOBV = 'N', V is not referenced.
- **LDV (input)**

The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (input/output)**
  On entry, if JOBQ = 'Q', Q must contain a matrix Q1 (usually the orthogonal matrix returned by SGGSVP). On exit, if JOBQ = 'I', Q contains the orthogonal matrix Q; if JOBQ = 'Q', Q contains the product Q1*Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **NCYCLE (output)**
  The number of cycles required for convergence.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.

= 1:  the procedure does not converge after MAXIT cycles.
```

NAME

stgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z

SYNOPSIS

```
 SUBROUTINE STGSNA( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL, LDVL,
*      VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 REAL A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), S(*), DIF(*), WORK(*)

 SUBROUTINE STGSNA_64( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 REAL A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), S(*), DIF(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSNA( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, DIF, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR

 SUBROUTINE TGSNA_64( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB],
```

```
*       VL, [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, DIF, WORK
 REAL, DIMENSION(:,:) :: A, B, VL, VR
```

C INTERFACE

#include <sunperf.h>

void stgsna(char job, char howmnt, logical *select, int n, float *a, int lda, float *b, int ldb, float *vl, int ldvl, float *vr, int ldvr, float *s, float *dif, int mm, int *m, int *info);

void stgsna_64(char job, char howmnt, logical *select, long n, float *a, long lda, float *b, long ldb, float *vl, long ldvl, float *vr, long ldvr, float *s, float *dif, long mm, long *m, long *info);

PURPOSE

stgsna estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair (Q*A*Z', Q*B*Z') with orthogonal matrices Q and Z, where Z' denotes the transpose of Z.

(A, B) must be in generalized real Schur form (as returned by SGGES), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (DIF):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (DIF);

  = 'B': for both eigenvalues and eigenvectors (S and DIF).
  ```
- **HOWMNT (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNT = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the eigenpair corresponding to a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select condition numbers corresponding to a complex conjugate pair of eigenvalues `w(j)` and w(j+1), either `SELECT(j)`

or SELECT(j+1) or both, must be set to .TRUE.. If HOWMNT = 'A', SELECT is not referenced.

- **N (input)**
  The order of the square matrix pair (A, B). N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A in the pair (A,B).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The upper triangular matrix B in the pair (A,B).
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by STGEVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1. If JOB = 'E' or 'B', LDVL >= N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns ov VR, as returned by STGEVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1. If JOB = 'E' or 'B', LDVR >= N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of S are set to the same value. Thus S(j), DIF(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **DIF (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of DIF are set to the same value. If the eigenvalues cannot be reordered to compute DIF(j), DIF(j) is set to 0; this can only occur when the true value would be very small anyway. If JOB = 'E', DIF is not referenced.
- **MM (input)**
  The number of elements in the arrays S and DIF. MM >= M.
- **M (output)**
  The number of elements of the arrays S and DIF used to store the specified condition numbers; for each selected real eigenvalue one element is used, and for each selected complex conjugate pair of eigenvalues, two elements are used. If HOWMNT = 'A', M is set to N.
- **WORK (workspace)**
  If JOB = 'E', WORK is not referenced. Otherwise, on exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= N. If JOB = 'V' or 'B' LWORK >= 2*N*(N+2)+16.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  dimension(N+6) If JOB = 'E', IWORK is not referenced.
- **INFO (output)**

  =0: Successful exit

```
<0: If INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of a generalized eigenvalue w = (a, b) is defined as

(w) = (|u'Av|**2 + |u'Bv|**2)**(1/2) / (norm(u)*norm(v))

where u and v are the left and right eigenvectors of (A, B) corresponding to w; |z| denotes the absolute value of the complex number, and `norm(u)` denotes the 2-norm of the vector u.

The pair (a, b) corresponds to an eigenvalue w = a/b (= u'Av/u'Bv) of the matrix pair (A, B). If both a and b equal zero, then (A B) is singular and `S(I)` = -1 is returned.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(A, B) / `S(I)`

where EPS is the machine precision.

The reciprocal of the condition number `DIF(i)` of right eigenvector u and left eigenvector v corresponding to the generalized eigenvalue w is defined as follows:

a) If the i-th eigenvalue w = (a,b) is real

```
   Suppose U and V are orthogonal transformations such that

              U'*(A, B)*V   = (S, T)  = ( a    *  ) ( b  *  )  1
                                        ( 0  S22 ),( 0 T22 )  n-1
                                          1  n-1     1 n-1

   Then the reciprocal condition number DIF(i) is

              Difl((a, b), (S22, T22))  = sigma-min( Zl ),

   where sigma-min(Zl) denotes the smallest singular value of the
   2(n-1)-by-2(n-1) matrix

       Zl  = [ kron(a, In-1)  -kron(1, S22) ]

             [ kron(b, In-1)  -kron(1, T22) ] .

   Here In-1 is the identity matrix of size n-1. kron(X, Y) is the
   Kronecker product between the matrices X and Y.

   Note that if the default method for computing DIF(i) is wanted
   (see SLATDF), then the parameter DIFDRI (see below) should be
   changed from 3 to 4 (routine SLATDF(IJOB  = 2 will be used)).
   See STGSYL for more details.
```

b) If the i-th and (i+1)-th eigenvalues are complex conjugate pair,

```
Suppose U and V are orthogonal transformations such that

           U'*(A, B)*V  = (S, T)  = ( S11  *   ) ( T11  *  )  2
                                    ( 0    S22 ),( 0    T22) n-2
                                      2    n-2     2    n-2

and (S11, T11) corresponds to the complex conjugate eigenvalue
pair (w, conjg(w)). There exist unitary matrices U1 and V1 such
that

    U1'*S11*V1  = ( s11 s12 )   and U1'*T11*V1  = ( t11 t12 )
                  (  0  s22 )                     (  0  t22 )

where the generalized eigenvalues w  = s11/t11 and

conjg(w)  = s22/t22.

Then the reciprocal condition number DIF(i) is bounded by

    min( d1, max( 1, |real(s11)/real(s22)| )*d2 )

where, d1  = Difl((s11, t11), (s22, t22))  = sigma-min(Z1), where
Z1 is the complex 2-by-2 matrix

         Z1  =  [ s11  -s22 ]

                [ t11  -t22 ],

This is done by computing (using real arithmetic) the

roots of the characteristical polynomial det(Z1' * Z1 - lambda I),
where Z1' denotes the conjugate transpose of Z1 and det(X) denotes
the determinant of X.

and d2 is an upper bound on Difl((S11, T11), (S22, T22)), i.e. an
upper bound on sigma-min(Z2), where Z2 is (2n-2)-by-(2n-2)

         Z2  = [ kron(S11', In-2)  -kron(I2, S22) ]

               [ kron(T11', In-2)  -kron(I2, T22) ]

Note that if the default method for computing DIF is wanted (see
SLATDF), then the parameter DIFDRI (see below) should be changed
from 3 to 4 (routine SLATDF(IJOB  = 2 will be used)). See STGSYL
for more details.
```

For each eigenvalue/vector specified by SELECT, DIF stores a Frobenius norm-based estimate of Difl.

An approximate error bound for the i-th computed eigenvector VL(i) or VR(i) is given by

```
          EPS * norm(A, B) / DIF(i).
```

See ref. [2-3] for more details and further references.

Based on contributions by

Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.

References

= = = = = = = = = =

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software,

Report UMINF - 94.04, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87. To appear in Numerical Algorithms, 1996.

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

stgsyl - solve the generalized Sylvester equation

SYNOPSIS

```
 SUBROUTINE STGSYL( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER IWORK(*)
 REAL SCALE, DIF
 REAL A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)

 SUBROUTINE STGSYL_64( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 INTEGER*8 IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER*8 IWORK(*)
 REAL SCALE, DIF
 REAL A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSYL( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C, [LDC],
*       D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: TRANS
 INTEGER :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL :: SCALE, DIF
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A, B, C, D, E, F

 SUBROUTINE TGSYL_64( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C,
*       [LDC], D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK],
*       [IWORK], [INFO])
```

```
CHARACTER(LEN=1) :: TRANS
INTEGER(8) :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL :: SCALE, DIF
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A, B, C, D, E, F
```

C INTERFACE

#include <sunperf.h>

void stgsyl(char trans, int ijob, int m, int n, float *a, int lda, float *b, int ldb, float *c, int ldc, float *d, int ldd, float *e, int lde, float *f, int ldf, float *scale, float *dif, int *info);

void stgsyl_64(char trans, long ijob, long m, long n, float *a, long lda, float *b, long ldb, float *c, long ldc, float *d, long ldd, float *e, long lde, float *f, long ldf, float *scale, float *dif, long *info);

PURPOSE

stgsyl solves the generalized Sylvester equation:

```
A * R - L * B = scale * C                (1)
D * R - L * E = scale * F
```

where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively, with real entries. (A, D) and (B, E) must be in generalized (real) Schur canonical form, i.e. A, B are upper quasi triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F). 0 <= SCALE <= 1 is an output scaling factor chosen to avoid overflow.

In matrix notation (1) is equivalent to solve Zx = scale b, where Z is defined as

```
Z = [ kron(In, A)  -kron(B', Im) ]         (2)
    [ kron(In, D)  -kron(E', Im) ].
```

Here Ik is the identity matrix of size k and X' is the transpose of X. kron(X, Y) is the Kronecker product between the matrices X and Y.

If TRANS = 'T', STGSYL solves the transposed system Z'*y = scale*b, which is equivalent to solve for R and L in

```
A' * R  + D' * L   = scale *  C          (3)
R  * B' + L  * E'  = scale * (-F)
```

This case (TRANS = 'T') is used to compute an one-norm-based estimate of Dif[(A,D), (B,E)], the separation between the matrix pairs (A,D) and (B,E), using SLACON.

If IJOB >= 1, STGSYL computes a Frobenius norm-based estimate of Dif[(A,D),(B,E)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z. See [1-2] for more information.

This is a level 3 BLAS algorithm.

ARGUMENTS

- **TRANS (input)**

```
 = 'N', solve the generalized Sylvester equation (1).
 = 'T', solve the 'transposed' system (3).
```

- **IJOB (input)**
  Specifies what kind of functionality to be performed. =0: solve (1) only.

```
 =1: The functionality of 0 and 3.

 =2: The functionality of 0 and 4.

 =3: Only an estimate of Dif[(A,D), (B,E)] is computed.
(look ahead strategy IJOB   = 1 is used).
 =4: Only an estimate of Dif[(A,D), (B,E)] is computed.
( SGECON on sub-systems is used ).
Not referenced if TRANS  = 'T'.
```

- **M (input)**
  The order of the matrices A and D, and the row dimension of the matrices C, F, R and L.
- **N (input)**
  The order of the matrices B and E, and the column dimension of the matrices C, F, R and L.
- **A (input)**
  The upper quasi triangular matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, M).
- **B (input)**
  The upper quasi triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **C (input/output)**
  On entry, C contains the right-hand-side of the first matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, C has been overwritten by the solution R. If IJOB = 3 or 4 and TRANS = 'N', C holds R, the solution achieved during the computation of the Dif-estimate.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1, M).
- **D (input)**
  The upper triangular matrix D.
- **LDD (input)**
  The leading dimension of the array D. LDD > = max(1, M).
- **E (input)**
  The upper triangular matrix E.
- **LDE (input)**
  The leading dimension of the array E. LDE > = max(1, N).
- **F (input/output)**
  On entry, F contains the right-hand-side of the second matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, F has been overwritten by the solution L. If IJOB = 3 or 4 and TRANS = 'N', F holds L, the solution achieved during the computation of the Dif-estimate.
- **LDF (input)**
  The leading dimension of the array F. LDF > = max(1, M).
- **SCALE (output)**

On exit DIF is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. DIF is an upper bound of Dif[(A,D), (B,E)] = sigma_min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'T', DIF is not touched.

- **DIF (output)**
  On exit DIF is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. DIF is an upper bound of Dif[(A,D), (B,E)] = sigma_min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'T', DIF is not touched.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. If IJOB = 1 or 2 and TRANS = 'N', LWORK >= 2*M*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  `dimension(M+N+2)`
- **INFO (output)**

```
=0: successful exit

<0: If INFO  = -i, the i-th argument had an illegal value.

>0: (A, D) and (B, E) have common or close eigenvalues.
```

FURTHER DETAILS

Based on contributions by

```
Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

[2] B. Kagstrom, A Perturbation Analysis of the Generalized Sylvester Equation (AR - LB, DR - LE) = (C, F), SIAM J. Matrix Anal. Appl., 15(4):1045-1060, 1994

[3] B. Kagstrom and L. Westin, Generalized Schur Methods with Condition Estimators for Solving the Generalized Sylvester Equation, IEEE Transactions on Automatic Control, Vol. 34, No. 7, July 1989, pp 745-751.

NAME

stpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
  SUBROUTINE STPCON( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2, INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  INTEGER N, INFO
  INTEGER WORK2(*)
  REAL RCOND
  REAL A(*), WORK(*)

  SUBROUTINE STPCON_64( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2,
 *      INFO)
  CHARACTER * 1 NORM, UPLO, DIAG
  INTEGER*8 N, INFO
  INTEGER*8 WORK2(*)
  REAL RCOND
  REAL A(*), WORK(*)
```

F95 INTERFACE

```
  SUBROUTINE TPCON( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
 *       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  INTEGER :: N, INFO
  INTEGER, DIMENSION(:) :: WORK2
  REAL :: RCOND
  REAL, DIMENSION(:) :: A, WORK

  SUBROUTINE TPCON_64( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
 *       [INFO])
  CHARACTER(LEN=1) :: NORM, UPLO, DIAG
  INTEGER(8) :: N, INFO
  INTEGER(8), DIMENSION(:) :: WORK2
  REAL :: RCOND
  REAL, DIMENSION(:) :: A, WORK
```

C INTERFACE

#include <sunperf.h>

void stpcon(char norm, char uplo, char diag, int n, float *a, float *rcond, int *info);

void stpcon_64(char norm, char uplo, char diag, long n, float *a, float *rcond, long *info);

PURPOSE

stpcon estimates the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

stpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE STPMV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, INCY
REAL A(*), Y(*)

SUBROUTINE STPMV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, INCY
REAL A(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TPMV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, INCY
REAL, DIMENSION(:) :: A, Y

SUBROUTINE TPMV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, INCY
REAL, DIMENSION(:) :: A, Y
```

C INTERFACE

#include <sunperf.h>

void stpmv(char uplo, char transa, char diag, int n, float *a, float *y, int incy);

void stpmv_64(char uplo, char transa, char diag, long n, float *a, float *y, long incy);

PURPOSE

stpmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

stprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE STPRFS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE STPRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TPRFS( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB], X,
*        [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: A, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X

 SUBROUTINE TPRFS_64( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: A, FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: B, X
```

C INTERFACE

#include <sunperf.h>

void stprfs(char uplo, char transa, char diag, int n, int nrhs, float *a, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void stprfs_64(char uplo, char transa, char diag, long n, long nrhs, float *a, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

stprfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix.

The solution matrix X must be computed by STPTRS or some other means before entering this routine. STPRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

stpsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE STPSV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, INCY
REAL A(*), Y(*)

SUBROUTINE STPSV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, INCY
REAL A(*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TPSV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, INCY
REAL, DIMENSION(:) :: A, Y

SUBROUTINE TPSV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, INCY
REAL, DIMENSION(:) :: A, Y
```

C INTERFACE

#include <sunperf.h>

void stpsv(char uplo, char transa, char diag, int n, float *a, float *y, int incy);

void stpsv_64(char uplo, char transa, char diag, long n, float *a, float *y, long incy);

PURPOSE

stpsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

stptri - compute the inverse of a real upper or lower triangular matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE STPTRI( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, INFO
REAL A(*)

SUBROUTINE STPTRI_64( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, INFO
REAL A(*)
```

F95 INTERFACE

```
SUBROUTINE TPTRI( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, INFO
REAL, DIMENSION(:) :: A

SUBROUTINE TPTRI_64( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, INFO
REAL, DIMENSION(:) :: A
```

C INTERFACE

#include <sunperf.h>

void stptri(char uplo, char diag, int n, float *a, int *info);

void stptri_64(char uplo, char diag, long n, float *a, long *info);

PURPOSE

stptri computes the inverse of a real upper or lower triangular matrix A stored in packed format.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangular matrix A, stored columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*((2*n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details. On exit, the (triangular) inverse of the original matrix, in the same packed storage format.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

FURTHER DETAILS

A triangular matrix A can be transferred to packed storage using one of the following program segments:

UPLO = 'U': UPLO = 'L':

```
JC  = 1                          JC  = 1

DO 2 J  = 1, N                   DO 2 J  = 1, N
```

```
     DO 1 I  = 1, J
        A(JC+I-1)  = A(I,J)
1    CONTINUE
     JC  = JC + J
2 CONTINUE
```

```
      DO 1 I  = J, N
        A(JC+I-J)  = A(I,J)
1    CONTINUE
      JC  = JC + N - J + 1
2 CONTINUE
```

NAME

stptrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
SUBROUTINE STPTRS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)

SUBROUTINE STPTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, NRHS, LDB, INFO
REAL A(*), B(LDB,*)
```

F95 INTERFACE

```
SUBROUTINE TPTRS( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B

SUBROUTINE TPTRS_64( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, NRHS, LDB, INFO
REAL, DIMENSION(:) :: A
REAL, DIMENSION(:,:) :: B
```

C INTERFACE

#include <sunperf.h>

void stptrs(char uplo, char transa, char diag, int n, int nrhs, float *a, float *b, int ldb, int *info);

void stptrs_64(char uplo, char transa, char diag, long n, long nrhs, float *a, float *b, long ldb, long *info);

PURPOSE

stptrs solves a triangular system of the form

where A is a triangular matrix of order N stored in packed format, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B  (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

 < 0: if INFO = -i, the i-th argument had an illegal value

 > 0: if INFO = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.

NAME

strans - transpose and scale source matrix

SYNOPSIS

```
SUBROUTINE STRANS( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
INTEGER M, N
REAL SCALE
REAL SOURCE(*), DEST(*)

SUBROUTINE STRANS_64( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
INTEGER*8 M, N
REAL SCALE
REAL SOURCE(*), DEST(*)
```

F95 INTERFACE

```
SUBROUTINE TRANS( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
INTEGER :: M, N
REAL :: SCALE
REAL, DIMENSION(:) :: SOURCE, DEST

SUBROUTINE TRANS_64( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
INTEGER(8) :: M, N
REAL :: SCALE
REAL, DIMENSION(:) :: SOURCE, DEST
```

C INTERFACE

#include <sunperf.h>

void strans(char place, float scale, float *source, int m, int n, float *dest);

void strans_64(char place, float scale, float *source, long m, long n, float *dest);

PURPOSE

strans scales and transposes the source matrix. The N2 x N1 result is written into SOURCE when PLACE = 'I' or 'i', and DEST when PLACE = 'O' or 'o'.

```
PLACE = 'I' or 'i': SOURCE = SCALE * SOURCE'

PLACE = 'O' or 'o': DEST = SCALE * SOURCE'
```

ARGUMENTS

- **PLACE (input)**
  Type of transpose. 'I' or 'i' for in-place, 'O' or 'o' for out-of-place. 'I' is default.
- **SCALE (input)**
  Scale factor on the SOURCE matrix.
- **SOURCE (input/output)**
  (M, N) on input. Array of (N, M) on output if in-place transpose.
- **M (input)**
  Number of rows in the SOURCE matrix on input.
- **N (input)**
  Number of columns in the SOURCE matrix on input.
- **DEST (output)**
  Scaled and transposed SOURCE matrix if out-of-place transpose. Not referenced if in-place transpose.

NAME

strcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE STRCON( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER N, LDA, INFO
 INTEGER WORK2(*)
 REAL RCOND
 REAL A(LDA,*), WORK(*)

 SUBROUTINE STRCON_64( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 INTEGER*8 N, LDA, INFO
 INTEGER*8 WORK2(*)
 REAL RCOND
 REAL A(LDA,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRCON( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL :: RCOND
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: A

 SUBROUTINE TRCON_64( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 INTEGER(8) :: N, LDA, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL :: RCOND
```

```
REAL, DIMENSION(:) :: WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void strcon(char norm, char uplo, char diag, int n, float *a, int lda, float *rcond, int *info);

void strcon_64(char norm, char uplo, char diag, long n, float *a, long lda, float *rcond, long *info);

PURPOSE

strcon estimates the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

strevc - compute some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T

SYNOPSIS

```
 SUBROUTINE STREVC( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 REAL T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)

 SUBROUTINE STREVC_64( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 REAL T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREVC( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*        VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: T, VL, VR

 SUBROUTINE TREVC_64( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*        VR, [LDVR], MM, M, [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: T, VL, VR
```

C INTERFACE

#include <sunperf.h>

void strevc(char side, char howmny, logical *select, int n, float *t, int ldt, float *vl, int ldvl, float *vr, int ldvr, int mm, int *m, int *info);

void strevc_64(char side, char howmny, logical *select, long n, float *t, long ldt, float *vl, long ldvl, float *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

strevc computes some or all of the right and/or left eigenvectors of a real upper quasi-triangular matrix T.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:

```
T*x = w*x,     y'*T = w*y'
```

where y' denotes the conjugate transpose of the vector y.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of T, or the products Q*X and/or Q*Y, where Q is an input orthogonal

matrix. If T was obtained from the real-Schur factorization of an original matrix A = Q*T*Q', then Q*X and Q*Y are the matrices of right or left eigenvectors of A.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign. Corresponding to each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R':  compute right eigenvectors only;

  = 'L':  compute left eigenvectors only;

  = 'B':  compute both right and left eigenvectors.
  ```

- **HOWMNY (input)**

  ```
  = 'A':  compute all right and/or left eigenvectors;

  = 'B':  compute all right and/or left eigenvectors,
and backtransform them using the input matrices
supplied in VR and/or VL;
 = 'S':  compute selected right and/or left eigenvectors,
  ```

specified by the logical array SELECT.

- **SELECT (input/output)**
  If HOWMNY = 'S', SELECT specifies the eigenvectors to be computed. If HOWMNY = 'A' or 'B', SELECT is not referenced. To select the real eigenvector corresponding to a real eigenvalue w(j), SELECT(j) must be set to .TRUE.. To select the complex eigenvector corresponding to a complex conjugate pair w(j) and w(j+1), either SELECT(j) or SELECT(j+1) must be set to .TRUE.; then on exit SELECT(j) is .TRUE. and SELECT(j+1) is .FALSE..
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  The upper quasi-triangular matrix T in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the orthogonal matrix Q of Schur vectors returned by SHSEQR). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of T; VL has the same quasi-lower triangular form as T'. If T(i,i) is a real eigenvalue, then the i-th column VL(i) of VL is its corresponding eigenvector. If T(i:i+1,i:i+1) is a 2-by-2 block whose eigenvalues are complex-conjugate eigenvalues of T, then VL(i)+sqrt(-1)*VL(i+1) is the complex eigenvector corresponding to the eigenvalue with positive real part. if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of T specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= max(1,N) if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the orthogonal matrix Q of Schur vectors returned by SHSEQR). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of T; VR has the same quasi-upper triangular form as T. If T(i,i) is a real eigenvalue, then the i-th column VR(i) of VR is its corresponding eigenvector. If T(i:i+1,i:i+1) is a 2-by-2 block whose eigenvalues are complex-conjugate eigenvalues of T, then VR(i)+sqrt(-1)*VR(i+1) is the complex eigenvector corresponding to the eigenvalue with positive real part. if HOWMNY = 'B', the matrix Q*X; if HOWMNY = 'S', the right eigenvectors of T specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part and the second the imaginary part. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= max(1,N) if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
- **WORK (workspace)**
  dimension(3*N)
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

FURTHER DETAILS

The algorithm used in this program is basically backward (forward) substitution, with scaling to make the the code robust against possible overflow.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x| + |y|.

NAME

strexc - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST

SYNOPSIS

```
 SUBROUTINE STREXC( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, WORK, INFO)
 CHARACTER * 1 COMPQ
 INTEGER N, LDT, LDQ, IFST, ILST, INFO
 REAL T(LDT,*), Q(LDQ,*), WORK(*)

 SUBROUTINE STREXC_64( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, WORK,
*      INFO)
 CHARACTER * 1 COMPQ
 INTEGER*8 N, LDT, LDQ, IFST, ILST, INFO
 REAL T(LDT,*), Q(LDQ,*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREXC( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: COMPQ
 INTEGER :: N, LDT, LDQ, IFST, ILST, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: T, Q

 SUBROUTINE TREXC_64( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: COMPQ
 INTEGER(8) :: N, LDT, LDQ, IFST, ILST, INFO
 REAL, DIMENSION(:) :: WORK
 REAL, DIMENSION(:,:) :: T, Q
```

C INTERFACE

#include <sunperf.h>

void strexc(char compq, int n, float *t, int ldt, float *q, int ldq, int *ifst, int *ilst, int *info);

void strexc_64(char compq, long n, float *t, long ldt, float *q, long ldq, long *ifst, long *ilst, long *info);

PURPOSE

strexc reorders the real Schur factorization of a real matrix A = Q*T*Q**T, so that the diagonal block of T with row index IFST is moved to row ILST.

The real Schur form T is reordered by an orthogonal similarity transformation Z**T*T*Z, and optionally the matrix Q of Schur vectors is updated by postmultiplying it with Z.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **COMPQ (input)**

  ```
  = 'V':  update the matrix Q of Schur vectors;

  = 'N':  do not update Q.
  ```

- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input/output)**
  On entry, the upper quasi-triangular matrix T, in Schur Schur canonical form. On exit, the reordered upper quasi-triangular matrix, again in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the orthogonal transformation matrix Z which reorders T. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of T. The block with row index IFST is moved to row ILST, by a sequence of transpositions between adjacent blocks. On exit, if IFST pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; ILST always points to the first row of the block in its final position (which may differ from its input value by +1 or -1). 1 < = IFST < = N; 1 < = ILST < = N.
- **ILST (input/output)**
  See the description of IFST.
- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 = 1:  two adjacent blocks were too close to swap (the problem
is very ill-conditioned); T may have been partially
reordered, and ILST points to the first row of the
current position of the block being moved.
```

NAME

strmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A)

SYNOPSIS

```
 SUBROUTINE STRMM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER M, N, LDA, LDB
 REAL ALPHA
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE STRMM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER*8 M, N, LDA, LDB
 REAL ALPHA
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRMM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER :: M, N, LDA, LDB
 REAL :: ALPHA
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE TRMM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER(8) :: M, N, LDA, LDB
 REAL :: ALPHA
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void strmm(char side, char uplo, char transa, char diag, int m, int n, float alpha, float *a, int lda, float *b, int ldb);

void strmm_64(char side, char uplo, char transa, char diag, long m, long n, float alpha, float *a, long lda, float *b, long ldb);

PURPOSE

strmm performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'.
```

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) multiplies B from the left or right as follows:

  SIDE = 'L' or 'l' B : = alpha*op(A)*B.

  SIDE = 'R' or 'r' B : = alpha*B*op(A).

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = A'.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be one. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), when SIDE = 'R' or 'r' then LDA > = max(1, n). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading m by n part of the array B must contain the matrix B, and on exit is overwritten by the transformed matrix.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.

NAME

strmv - perform one of the matrix-vector operations x := A*x, or x := A'*x

---

SYNOPSIS

```
SUBROUTINE STRMV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, LDA, INCY
REAL A(LDA,*), Y(*)

SUBROUTINE STRMV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, LDA, INCY
REAL A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TRMV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE TRMV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void strmv(char uplo, char transa, char diag, int n, float *a, int lda, float *y, int incy);

void strmv_64(char uplo, char transa, char diag, long n, float *a, long lda, float *y, long incy);

PURPOSE

strmv performs one of the matrix-vector operations x := A*x, or x := A'*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = A'*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

strrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE STRRFS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB, X,
*      LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDA, LDB, LDX, INFO
 INTEGER WORK2(*)
 REAL A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)

 SUBROUTINE STRRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDA, LDB, LDX, INFO
 INTEGER*8 WORK2(*)
 REAL A(LDA,*), B(LDB,*), X(LDX,*), FERR(*), BERR(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRRFS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDA, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, B, X

 SUBROUTINE TRRFS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDA, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: WORK2
 REAL, DIMENSION(:) :: FERR, BERR, WORK
 REAL, DIMENSION(:,:) :: A, B, X
```

C INTERFACE

#include <sunperf.h>

void strrfs(char uplo, char transa, char diag, int n, int nrhs, float *a, int lda, float *b, int ldb, float *x, int ldx, float *ferr, float *berr, int *info);

void strrfs_64(char uplo, char transa, char diag, long n, long nrhs, float *a, long lda, float *b, long ldb, float *x, long ldx, float *ferr, float *berr, long *info);

PURPOSE

strrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix.

The solution matrix X must be computed by STRTRS or some other means before entering this routine. STRRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is

not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(3*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

strsen - reorder the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T,

---

SYNOPSIS

```
 SUBROUTINE STRSEN( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, WR, WI, M,
*       S, SEP, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 INTEGER N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 REAL S, SEP
 REAL T(LDT,*), Q(LDQ,*), WR(*), WI(*), WORK(*)

 SUBROUTINE STRSEN_64( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, WR, WI,
*       M, S, SEP, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 INTEGER*8 N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 REAL S, SEP
 REAL T(LDT,*), Q(LDQ,*), WR(*), WI(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TRSEN( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], WR,
*        WI, M, S, SEP, WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 INTEGER :: N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL :: S, SEP
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: T, Q
```

```
 SUBROUTINE TRSEN_64( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ],
*       WR, WI, M, S, SEP, WORK, [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 INTEGER(8) :: N, LDT, LDQ, M, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL :: S, SEP
 REAL, DIMENSION(:) :: WR, WI, WORK
 REAL, DIMENSION(:,:) :: T, Q
```

C INTERFACE

#include <sunperf.h>

void strsen(char job, char compq, logical *select, int n, float *t, int ldt, float *q, int ldq, float *wr, float *wi, int *m, float *s, float *sep, float *work, int lwork, int *info);

void strsen_64(char job, char compq, logical *select, long n, float *t, long ldt, float *q, long ldq, float *wr, float *wi, long *m, float *s, float *sep, float *work, long lwork, long *info);

PURPOSE

strsen reorders the real Schur factorization of a real matrix A = Q*T*Q**T, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elemnts equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (S) or the invariant subspace (SEP):

  ```
  = 'N': none;

  = 'E': for eigenvalues only (S);

  = 'V': for invariant subspace only (SEP);

  = 'B': for both eigenvalues and invariant subspace (S and
  SEP).
  ```

- **COMPQ (input)**

  ```
  = 'V': update the matrix Q of Schur vectors;
  ```

= 'N': do not update Q.

- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select a real eigenvalue w(j), SELECT(j) must be set to .TRUE.. To select a complex conjugate pair of eigenvalues w(j) and w(j+1), corresponding to a 2-by-2 diagonal block, either SELECT(j) or SELECT(j+1) or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  On entry, the upper quasi-triangular matrix T, in Schur canonical form. On exit, T is overwritten by the reordered matrix T, again in Schur canonical form, with the selected eigenvalues in the leading diagonal blocks.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the orthogonal transformation matrix which reorders T; the leading M columns of Q form an orthonormal basis for the specified invariant subspace. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; and if COMPQ = 'V', LDQ >= N.
- **WR (output)**
  The real and imaginary parts, respectively, of the reordered eigenvalues of T. The eigenvalues are stored in the same order as on the diagonal of T, with WR(i) = T(i,i) and, if T(i:i+1,i:i+1) is a 2-by-2 diagonal block, WI(i) > 0 and WI(i+1) = -WI(i). Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
- **WI (output)**
  See the description of WR.
- **M (output)**
  The dimension of the specified invariant subspace. 0 <= M <= N.
- **S (output)**
  If JOB = 'E' or 'B', S is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. S cannot underestimate the true reciprocal condition number by more than a factor of sqrt(N). If M = 0 or N, S = 1. If JOB = 'N' or 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', SEP is the estimated reciprocal condition number of the specified invariant subspace. If M = 0 or N, SEP = norm(T). If JOB = 'N' or 'E', SEP is not referenced.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOB = 'N', LWORK >= max(1,N); if JOB = 'E', LWORK >= M*(N-M); if JOB = 'V' or 'B', LWORK >= 2*M*(N-M).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If JOB = 'N' or 'E', IWORK is not referenced.
- **LIWORK (input)**
  The dimension of the array IWORK. If JOB = 'N' or 'E', LIWORK >= 1; if JOB = 'V' or 'B', LIWORK >= M*(N-M).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1: reordering of T failed because some eigenvalues are too
close to separate (the problem is very ill-conditioned);
T may have been partially reordered, and WR and WI contain
the eigenvalues in the same order as in T; S and SEP
(if requested) are set to zero.
```

FURTHER DETAILS

STRSEN first collects the selected eigenvalues by computing an orthogonal transformation Z to move them to the top left corner of T. In other words, the selected eigenvalues are the eigenvalues of T11 in:

```
            Z'*T*Z  = ( T11 T12 ) n1

                      (  0  T22 ) n2

                         n1  n2
```

where N = n1+n2 and Z' means the transpose of Z. The first n1 columns of Z span the specified invariant subspace of T.

If T has been obtained from the real Schur factorization of a matrix A = Q*T*Q', then the reordered real Schur factorization of A is given by A = (Q*Z)*(Z'*T*Z)*(Q*Z)', and the first n1 columns of Q*Z span the corresponding invariant subspace of A.

The reciprocal condition number of the average of the eigenvalues of T11 may be returned in S. S lies between 0 (very badly conditioned) and 1 (very well conditioned). It is computed as follows. First we compute R so that

```
                      P  = ( I  R ) n1

                           ( 0  0 ) n2

                             n1 n2
```

is the projector on the invariant subspace associated with T11. R is the solution of the Sylvester equation:

```
                     T11*R - R*T22  = T12.
```

Let F-norm(M) denote the Frobenius-norm of M and 2-norm(M) denote the two-norm of M. Then S is computed as the lower bound

```
                   (1 + F-norm(R)**2)**(-1/2)
```

on the reciprocal of 2-norm(P), the true reciprocal condition number. S cannot underestimate 1 / 2-norm(P) by more than a factor of sqrt(N).

An approximate error bound for the computed average of the eigenvalues of T11 is

```
                      EPS * norm(T) / S
```

where EPS is the machine precision.

The reciprocal condition number of the right invariant subspace spanned by the first n1 columns of Z (or of Q*Z) is returned in SEP. SEP is defined as the separation of T11 and T22:

```
sep( T11, T22 )  = sigma-min( C )
```

where sigma-min(C) is the smallest singular value of the

n1*n2-by-n1*n2 matrix

```
C   = kprod( I(n2), T11 ) - kprod( transpose(T22), I(n1) )
```

`I(m)` is an m by m identity matrix, and kprod denotes the Kronecker product. We estimate sigma-min(C) by the reciprocal of an estimate of the 1-norm of inverse(C). The true reciprocal 1-norm of `inverse(C)` cannot differ from sigma-min(C) by more than a factor of sqrt(n1*n2).

When SEP is small, small changes in T can cause large changes in the invariant subspace. An approximate bound on the maximum angular error in the computed right invariant subspace is

```
EPS * norm(T) / SEP
```

NAME

strsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

SYNOPSIS

```
 SUBROUTINE STRSM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER M, N, LDA, LDB
 REAL ALPHA
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE STRSM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 INTEGER*8 M, N, LDA, LDB
 REAL ALPHA
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER :: M, N, LDA, LDB
 REAL :: ALPHA
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE TRSM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 INTEGER(8) :: M, N, LDA, LDB
 REAL :: ALPHA
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void strsm(char side, char uplo, char transa, char diag, int m, int n, float alpha, float *a, int lda, float *b, int ldb);

void strsm_64(char side, char uplo, char transa, char diag, long m, long n, float alpha, float *a, long lda, float *b, long ldb);

PURPOSE

strsm solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'.
```

The matrix X is overwritten on B.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) appears on the left or right of X as follows:

  SIDE = 'L' or 'l' op(A)*X = alpha*B.

  SIDE = 'R' or 'r' X*op(A) = alpha*B.

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = A'.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be one. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), when SIDE = 'R' or 'r' then LDA > = max(1, n). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading m by n part of the array B must contain the right-hand side matrix B, and on exit is overwritten by the solution matrix X.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.

NAME

strsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal)

SYNOPSIS

```
 SUBROUTINE STRSNA( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 INTEGER N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER WORK1(*)
 LOGICAL SELECT(*)
 REAL T(LDT,*), VL(LDVL,*), VR(LDVR,*), S(*), SEP(*), WORK(LDWORK,*)

 SUBROUTINE STRSNA_64( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER*8 WORK1(*)
 LOGICAL*8 SELECT(*)
 REAL T(LDT,*), VL(LDVL,*), VR(LDVR,*), S(*), SEP(*), WORK(LDWORK,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSNA( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 INTEGER, DIMENSION(:) :: WORK1
 LOGICAL, DIMENSION(:) :: SELECT
 REAL, DIMENSION(:) :: S, SEP
 REAL, DIMENSION(:,:) :: T, VL, VR, WORK

 SUBROUTINE TRSNA_64( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
```

```
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: WORK1
LOGICAL(8), DIMENSION(:) :: SELECT
REAL, DIMENSION(:) :: S, SEP
REAL, DIMENSION(:,:) :: T, VL, VR, WORK
```

C INTERFACE

#include <sunperf.h>

void strsna(char job, char howmny, logical *select, int n, float *t, int ldt, float *vl, int ldvl, float *vr, int ldvr, float *s, float *sep, int mm, int *m, int ldwork, int *info);

void strsna_64(char job, char howmny, logical *select, long n, float *t, long ldt, float *vl, long ldvl, float *vr, long ldvr, float *s, float *sep, long mm, long *m, long ldwork, long *info);

PURPOSE

strsna estimates reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a real upper quasi-triangular matrix T (or of any matrix Q*T*Q**T with Q orthogonal).

T must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (SEP):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (SEP);

  = 'B': for both eigenvalues and eigenvectors (S and SEP).
  ```
- **HOWMNY (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNY = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the eigenpair corresponding to a real eigenvalue w(j), `SELECT(j)` must be set to .TRUE.. To select condition numbers corresponding to a complex conjugate pair of eigenvalues `w(j)` and w(j+1), either `SELECT(j)` or `SELECT(j+1)` or both, must be set to .TRUE.. If HOWMNY = 'A', SELECT is not referenced.
- **N (input)**
  The order of the matrix T. N >= 0.

- **T (input)**
  The upper quasi-triangular matrix T, in Schur canonical form.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of T (or of any Q*T*Q**T with Q orthogonal), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by SHSEIN or STREVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and if JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of T (or of any Q*T*Q**T with Q orthogonal), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by SHSEIN or STREVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; and if JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of S are set to the same value. Thus S(j), SEP(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of SEP are set to the same value. If the eigenvalues cannot be reordered to compute SEP(j), `SEP(j)` is set to 0; this can only occur when the true value would be very small anyway. If JOB = 'E', SEP is not referenced.
- **MM (input)**
  The number of elements in the arrays S (if JOB = 'E' or 'B') and/or SEP (if JOB = 'V' or 'B'). MM > = M.
- **M (output)**
  The number of elements of the arrays S and/or SEP actually used to store the estimated condition numbers. If HOWMNY = 'A', M is set to N.
- **WORK (workspace)**
  `dimension(LDWORK,N+1)` If JOB = 'E', WORK is not referenced.
- **LDWORK (input)**
  The leading dimension of the array WORK. LDWORK > = 1; and if JOB = 'V' or 'B', LDWORK > = N.
- **WORK1 (workspace)**
  `dimension(N)` If JOB = 'E', WORK1 is not referenced.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of an eigenvalue lambda is defined as

```
S(lambda)  = |v'*u| / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of T corresponding to lambda; v' denotes the conjugate-transpose of v, and `norm(u)` denotes the Euclidean norm. These reciprocal condition numbers always lie between zero (very badly conditioned) and one (very well conditioned). If n = 1, `S(lambda)` is defined to be 1.

An approximate error bound for a computed eigenvalue `W(i)` is given by

```
EPS * norm(T) / S(i)
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u corresponding to lambda is defined as follows. Suppose

```
T  = ( lambda  c  )

     (   0    T22 )
```

Then the reciprocal condition number is

```
SEP( lambda, T22 )  = sigma-min( T22 - lambda*I )
```

where sigma-min denotes the smallest singular value. We approximate the smallest singular value by the reciprocal of an estimate of the one-norm of the inverse of T22 - lambda*I. If n = 1, `SEP(1)` is defined to be abs(T(1,1)).

An approximate error bound for a computed right eigenvector `VR(i)` is given by

```
EPS * norm(T) / SEP(i)
```

NAME

strsv - solve one of the systems of equations A*x = b, or A'*x = b

---

SYNOPSIS

```
SUBROUTINE STRSV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER N, LDA, INCY
REAL A(LDA,*), Y(*)

SUBROUTINE STRSV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
INTEGER*8 N, LDA, INCY
REAL A(LDA,*), Y(*)
```

F95 INTERFACE

```
SUBROUTINE TRSV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER :: N, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A

SUBROUTINE TRSV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
INTEGER(8) :: N, LDA, INCY
REAL, DIMENSION(:) :: Y
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void strsv(char uplo, char transa, char diag, int n, float *a, int lda, float *y, int incy);

void strsv_64(char uplo, char transa, char diag, long n, float *a, long lda, float *y, long incy);

PURPOSE

strsv solves one of the systems of equations A*x = b, or A'*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' A'*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**

On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

strsyl - solve the real Sylvester matrix equation

SYNOPSIS

```
 SUBROUTINE STRSYL( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C, LDC,
*       SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 INTEGER ISGN, M, N, LDA, LDB, LDC, INFO
 REAL SCALE
 REAL A(LDA,*), B(LDB,*), C(LDC,*)

 SUBROUTINE STRSYL_64( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C,
*       LDC, SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 INTEGER*8 ISGN, M, N, LDA, LDB, LDC, INFO
 REAL SCALE
 REAL A(LDA,*), B(LDB,*), C(LDC,*)
```

F95 INTERFACE

```
 SUBROUTINE TRSYL( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*        C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 INTEGER :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL :: SCALE
 REAL, DIMENSION(:,:) :: A, B, C

 SUBROUTINE TRSYL_64( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*        C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 INTEGER(8) :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL :: SCALE
 REAL, DIMENSION(:,:) :: A, B, C
```

C INTERFACE

#include <sunperf.h>

void strsyl(char trana, char tranb, int isgn, int m, int n, float *a, int lda, float *b, int ldb, float *c, int ldc, float *scale, int *info);

void strsyl_64(char trana, char tranb, long isgn, long m, long n, float *a, long lda, float *b, long ldb, float *c, long ldc, float *scale, long *info);

PURPOSE

strsyl solves the real Sylvester matrix equation:

```
op(A)*X + X*op(B) = scale*C or

op(A)*X - X*op(B) = scale*C,
```

where `op(A)` = A or A**T, and A and B are both upper quasi- triangular. A is M-by-M and B is N-by-N; the right hand side C and the solution X are M-by-N; and scale is an output scale factor, set <= 1 to avoid overflow in X.

A and B must be in Schur canonical form (as returned by SHSEQR), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

ARGUMENTS

- **TRANA (input)**
  Specifies the option op(A):

  ```
  = 'N': op(A)  = A    (No transpose)

  = 'T': op(A)  = A**T (Transpose)

  = 'C': op(A)  = A**H (Conjugate transpose  = Transpose)
  ```
- **TRANB (input)**
  Specifies the option op(B):

  ```
  = 'N': op(B)  = B    (No transpose)

  = 'T': op(B)  = B**T (Transpose)

  = 'C': op(B)  = B**H (Conjugate transpose  = Transpose)
  ```
- **ISGN (input)**
  Specifies the sign in the equation:

  ```
  = +1: solve op(A)*X + X*op(B)  = scale*C
  ```

= -1: solve op(A)*X - X*op(B) = scale*C

- **M (input)**
  The order of the matrix A, and the number of rows in the matrices X and C. M >= 0.
- **N (input)**
  The order of the matrix B, and the number of columns in the matrices X and C. N >= 0.
- **A (input)**
  The upper quasi-triangular matrix A, in Schur canonical form.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input)**
  The upper quasi-triangular matrix B, in Schur canonical form.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **C (input/output)**
  On entry, the M-by-N right hand side matrix C. On exit, C is overwritten by the solution matrix X.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)`
- **SCALE (output)**
  The scale factor, scale, set <= 1 to avoid overflow in X.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

= 1: A and B have common or very close eigenvalues; perturbed
values were used to solve the equation (but the matrices
A and B are unchanged).

NAME

strti2 - compute the inverse of a real upper or lower triangular matrix

---

SYNOPSIS

```
SUBROUTINE STRTI2( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, LDA, INFO
REAL A(LDA,*)

SUBROUTINE STRTI2_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE TRTI2( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE TRTI2_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void strti2(char uplo, char diag, int n, float *a, int lda, int *info);

void strti2_64(char uplo, char diag, long n, float *a, long lda, long *info);

PURPOSE

strti2 computes the inverse of a real upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **DIAG (input)**
  Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular

  ```
  = 'U':  Unit triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading n by n upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

  On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

strtri - compute the inverse of a real upper or lower triangular matrix A

SYNOPSIS

```
SUBROUTINE STRTRI( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER N, LDA, INFO
REAL A(LDA,*)

SUBROUTINE STRTRI_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
INTEGER*8 N, LDA, INFO
REAL A(LDA,*)
```

F95 INTERFACE

```
SUBROUTINE TRTRI( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A

SUBROUTINE TRTRI_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
INTEGER(8) :: N, LDA, INFO
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void strtri(char uplo, char diag, int n, float *a, int lda, int *info);

void strtri_64(char uplo, char diag, long n, float *a, long lda, long *info);

PURPOSE

strtri computes the inverse of a real upper or lower triangular matrix A.

This is the Level 3 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1. On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value

> 0: if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

NAME

strtrs - solve a triangular system of the form A * X = B or A**T * X = B,

---

SYNOPSIS

```
 SUBROUTINE STRTRS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER N, NRHS, LDA, LDB, INFO
 REAL A(LDA,*), B(LDB,*)

 SUBROUTINE STRTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 INTEGER*8 N, NRHS, LDA, LDB, INFO
 REAL A(LDA,*), B(LDB,*)
```

F95 INTERFACE

```
 SUBROUTINE TRTRS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER :: N, NRHS, LDA, LDB, INFO
 REAL, DIMENSION(:,:) :: A, B

 SUBROUTINE TRTRS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 INTEGER(8) :: N, NRHS, LDA, LDB, INFO
 REAL, DIMENSION(:,:) :: A, B
```

C INTERFACE

#include <sunperf.h>

void strtrs(char uplo, char transa, char diag, int n, int nrhs, float *a, int lda, float *b, int ldb, int *info);

void strtrs_64(char uplo, char transa, char diag, long n, long nrhs, float *a, long lda, float *b, long ldb, long *info);

PURPOSE

strtrs solves a triangular system of the form

where A is a triangular matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B  (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose  = Transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**

On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the solutions
X have not been computed.
```

NAME

stzrqf - routine is deprecated and has been replaced by routine STZRZF

SYNOPSIS

```
SUBROUTINE STZRQF( M, N, A, LDA, TAU, INFO)
INTEGER M, N, LDA, INFO
REAL A(LDA,*), TAU(*)

SUBROUTINE STZRQF_64( M, N, A, LDA, TAU, INFO)
INTEGER*8 M, N, LDA, INFO
REAL A(LDA,*), TAU(*)
```

F95 INTERFACE

```
SUBROUTINE TZRQF( [M], [N], A, [LDA], TAU, [INFO])
INTEGER :: M, N, LDA, INFO
REAL, DIMENSION(:) :: TAU
REAL, DIMENSION(:,:) :: A

SUBROUTINE TZRQF_64( [M], [N], A, [LDA], TAU, [INFO])
INTEGER(8) :: M, N, LDA, INFO
REAL, DIMENSION(:) :: TAU
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void stzrqf(int m, int n, float *a, int lda, float *tau, int *info);

void stzrqf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

stzrqf routine is deprecated and has been replaced by routine STZRZF.

STZRQF reduces the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N orthogonal matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= M.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the orthogonal matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
Z( k )  = ( I     0   ),

          ( 0  T( k ) )
```

where

```
T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                             (   0    )
```

```
                                        ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

stzrzf - reduce the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations

---

SYNOPSIS

```
SUBROUTINE STZRZF( M, N, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER M, N, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)

SUBROUTINE STZRZF_64( M, N, A, LDA, TAU, WORK, LWORK, INFO)
INTEGER*8 M, N, LDA, LWORK, INFO
REAL A(LDA,*), TAU(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE TZRZF( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
INTEGER :: M, N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A

SUBROUTINE TZRZF_64( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
INTEGER(8) :: M, N, LDA, LWORK, INFO
REAL, DIMENSION(:) :: TAU, WORK
REAL, DIMENSION(:,:) :: A
```

C INTERFACE

#include <sunperf.h>

void stzrzf(int m, int n, float *a, int lda, float *tau, int *info);

void stzrzf_64(long m, long n, float *a, long lda, float *tau, long *info);

PURPOSE

stzrzf reduces the M-by-N (M<=N) real upper trapezoidal matrix A to upper triangular form by means of orthogonal transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N orthogonal matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the orthogonal matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

```
  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
   Z( k )  = ( I     0   ),

             ( 0  T( k ) )
```

where

```
   T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                                (   0    )
                                                ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

sunperf_version - gets library information .HP 1i SUBROUTINE SUNPERF_VERSION(VERSION, PATCH, UPDATE) .HP 1i INTEGER VERSION, PATCH, UPDATE .HP 1i

---

SYNOPSIS

```
SUBROUTINE SUNPERF_VERSION( VERSION, PATCH, UPDATE)
INTEGER VERSION, PATCH, UPDATE

SUBROUTINE SUNPERF_VERSION_64( VERSION, PATCH, UPDATE)
INTEGER*8 VERSION, PATCH, UPDATE
```

F95 INTERFACE

```
SUBROUTINE SUNPERF_VERSION( VERSION, PATCH, UPDATE)
INTEGER :: VERSION, PATCH, UPDATE

SUBROUTINE SUNPERF_VERSION_64( VERSION, PATCH, UPDATE)
INTEGER(8) :: VERSION, PATCH, UPDATE
```

C INTERFACE

#include <sunperf.h>

void sunperf_version(int *version, int *patch, int *update);

void sunperf_version_64(long *version, long *patch, long *update);

ARGUMENTS

- **VERSION (output)**
  Version number of library
- **PATCH (output)**
  Patch number of library
- **UPDATE (output)**
  Update number of library

NAME

swiener - perform Wiener deconvolution of two signals

SYNOPSIS

```
SUBROUTINE SWIENER( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER N_POINTS, ISW, IERR
REAL ACOR(*), XCOR(*), FLTR(*), EROP(*)

SUBROUTINE SWIENER_64( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER*8 N_POINTS, ISW, IERR
REAL ACOR(*), XCOR(*), FLTR(*), EROP(*)
```

F95 INTERFACE

```
SUBROUTINE WIENER( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER :: N_POINTS, ISW, IERR
REAL, DIMENSION(:) :: ACOR, XCOR, FLTR, EROP

SUBROUTINE WIENER_64( N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR)
INTEGER(8) :: N_POINTS, ISW, IERR
REAL, DIMENSION(:) :: ACOR, XCOR, FLTR, EROP
```

C INTERFACE

#include <sunperf.h>

void swiener(int n_points, float *acor, float *xcor, float *fltr, float *erop, int *isw, int *ierr);

void swiener_64(long n_points, float *acor, float *xcor, float *fltr, float *erop, long *isw, long *ierr);

PURPOSE

swiener performs Wiener deconvolution of two signals.

ARGUMENTS

- **N_POINTS (input)**
  On entry, the number of points in the input correlations. Unchanged on exit.
- **ACOR (input)**
  On entry, autocorrelation coefficients. Unchanged on exit.
- **XCOR (input)**
  On entry, cross-correlation coefficients. Unchanged on exit.
- **FLTR (output)**
  On exit, filter coefficients. Unchanged on exit.
- **EROP (input)**
  On exit, the prediction error.
- **ISW (input/output)**
  On entry, if ISW .EQ. 0 then perform spiking deconvolution, otherwise perform general deconvolution. Unchanged on exit.
- **IERR (input/output)**
  On exit, the deconvolution was successful iff IERR .EQ. 0, otherwise there was an error.

NAME

use_threads - set the upper bound on the number of threads that the calling thread wants used

---

SYNOPSIS

```
SUBROUTINE USE_THREADS( NTHREADS)
INTEGER NTHREADS

SUBROUTINE USE_THREADS_64( NTHREADS)
INTEGER*8 NTHREADS
```

F95 INTERFACE

```
SUBROUTINE USE_THREADS( NTHREADS)
INTEGER :: NTHREADS

SUBROUTINE USE_THREADS_64( NTHREADS)
INTEGER(8) :: NTHREADS
```

C INTERFACE

#include <sunperf.h>

void use_threads(int nthreads);

void use_threads_64(long nthreads);

PURPOSE

use_threads THREADS sets an upper bound on the number of threads that the calling thread wants used. Subsequent calls to this routine result in replacement of the previous Use number for the calling thread. This counts all threads working on the callers behalf, so if it passes 2 for NTHREADS and then calls some subroutine, there will be at most 1 additional thread started to do the computation. There is no restriction that the sum of all NTHREADS from USE_THREADS calls may not exceed the number of CPUs in a system.

ARGUMENTS

NAME

using_threads - returns the current Use number set by the USE_THREADS subroutine

---

SYNOPSIS

```
INTEGER FUNCTION USING_THREADS( )

INTEGER*8 FUNCTION USING_THREADS_64( )
```

F95 INTERFACE

```
INTEGER FUNCTION USING_THREADS( )

INTEGER(8) FUNCTION USING_THREADS_64( )
```

C INTERFACE

#include <sunperf.h>

int using_threads();

long using_threads_64();

PURPOSE

using_threads THREADS will return the current Use number from the USE_THREADS subroutine for the calling thread.

ARGUMENTS

NAME

vcfftb - compute a periodic sequence from its Fourier coefficients. The VCFFT operations are normalized, so a call of VCFFTF followed by a call of VCFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VCFFTB( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER M, N, MDIMX

SUBROUTINE VCFFTB_64( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER*8 M, N, MDIMX
```

F95 INTERFACE

```
SUBROUTINE FFTB( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX, DIMENSION(:) :: WSAVE
COMPLEX, DIMENSION(:,:) :: X, XT
INTEGER :: M, N, MDIMX

SUBROUTINE FFTB_64( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX, DIMENSION(:) :: WSAVE
COMPLEX, DIMENSION(:,:) :: X, XT
INTEGER(8) :: M, N, MDIMX
```

C INTERFACE

#include <sunperf.h>

void vcfftb(int m, int n, complex *x, complex *xt, int mdimx, char rowcol, complex *wsave);

void vcfftb_64(long m, long n, complex *x, complex *xt, long mdimx, char rowcol, complex *wsave);

ARGUMENTS

- **M (input)**
  If ROWCOL = 'R' or 'r', M is the number of sequences to be transformed. Otherwise, M is the length of the sequences to be transformed. M > = 0.
- **N (input)**
  If ROWCOL = 'R' or 'r', N is the length of the sequences to be transformed. Otherwise, N is the number of sequences to be transformed. N > = 0.
- **X (input/output)**
  On entry, if ROWCOL = 'R' or 'r' `X(MDIMX,N)` is an array whose first M rows contain the sequences to be transformed. Otherwise, `X(MDIMX,N)` contains data sequences of length M stored in N columns of X.
- **XT (input)**
  A work array. The size of this workspace depends on the number of threads that are used to execute this routine. There are various functions that can be used to determine the number of threads available (get_env, available_threads, etc). The appropriate amount, which is (number of threads * length of data sequences), can then be dynamically allocated for XT from the driver routine. If XT can only be allocated statically, then the size of XT should be (length of data sequences * number of sequences).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **ROWCOL (input)**
  Indicates whether to transform rows ('R' or 'r') or columns ('C' or 'c').
- **WSAVE (input)**
  On entry, an array of dimension (L2+15) or greater, where L2 = 2*M if ROWCOL = ('R' or 'r'). Otherwise, L2 = 2*N. WSAVE is initialized by VCFFTI.

NAME

vcfftf - compute the Fourier coefficients of a periodic sequence. The VCFFT operations are normalized, so a call of VCFFTF followed by a call of VCFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VCFFTF( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER M, N, MDIMX

SUBROUTINE VCFFTF_64( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER*8 M, N, MDIMX
```

F95 INTERFACE

```
SUBROUTINE FFTF( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX, DIMENSION(:) :: WSAVE
COMPLEX, DIMENSION(:,:) :: X, XT
INTEGER :: M, N, MDIMX

SUBROUTINE FFTF_64( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX, DIMENSION(:) :: WSAVE
COMPLEX, DIMENSION(:,:) :: X, XT
INTEGER(8) :: M, N, MDIMX
```

C INTERFACE

#include <sunperf.h>

void vcfftf(int m, int n, complex *x, complex *xt, int mdimx, char rowcol, complex *wsave);

void vcfftf_64(long m, long n, complex *x, complex *xt, long mdimx, char rowcol, complex *wsave);

ARGUMENTS

- **M (input)**
  If ROWCOL = 'R' or 'r', M is the number of sequences to be transformed. Otherwise, M is the length of the sequences to be transformed. M > = 0.
- **N (input)**
  If ROWCOL = 'R' or 'r', N is the length of the sequences to be transformed. Otherwise, N is the number of sequences to be transformed. N > = 0.
- **X (input/output)**
  On entry, if ROWCOL = 'R' or 'r' `X(MDIMX,N)` is an array whose first M rows contain the sequences to be transformed. Otherwise, `X(MDIMX,N)` contains data sequences of length M stored in N columns of X.
- **XT (input)**
  A work array. The size of this workspace depends on the number of threads that are used to execute this routine. There are various functions that can be used to determine the number of threads available (get_env, available_threads, etc). The appropriate amount, which is (number of threads * length of data sequences), can then be dynamically allocated for XT from the driver routine. If XT can only be allocated statically, then the size of XT should be (length of data sequences * number of sequences).
- **MDIMX (input)**
  Leading dimension of the arrays X. MDIMX > = M.
- **ROWCOL (input)**
  Indicates whether data sequences in X are stored row-wise ('R' or 'r') or column-wise ('C' or 'c').
- **WSAVE (input)**
  On entry, an array of dimension (L2+15) or greater, where L2 = 2*M if ROWCOL = ('R' or 'r'). Otherwise, L2 = 2*N. WSAVE is initialized by VCFFTI.

NAME

vcffti - initialize the array WSAVE, which is used in both VCFFTF and VCFFTB.

SYNOPSIS

```
SUBROUTINE VCFFTI( N, WSAVE)
COMPLEX WSAVE(*)
INTEGER N

SUBROUTINE VCFFTI_64( N, WSAVE)
COMPLEX WSAVE(*)
INTEGER*8 N
```

F95 INTERFACE

```
SUBROUTINE VFFTI( N, WSAVE)
COMPLEX, DIMENSION(:) :: WSAVE
INTEGER :: N

SUBROUTINE VFFTI_64( N, WSAVE)
COMPLEX, DIMENSION(:) :: WSAVE
INTEGER(8) :: N
```

C INTERFACE

#include <sunperf.h>

void vcffti(int n, complex *wsave);

void vcffti_64(long n, complex *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2*N + 15) or greater. VCFFTI needs to be called only once to initialize WSAVE before calling VCFFTF and/or VCFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VCOSQB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VCOSQB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE COSQB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vcosqb(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vcosqb_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, the rows contain the sequences to be transformed. On exit, the quarter-wave cosine synthesis of the input.
- **XT (input)**
  A work array.
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VCOSQI.

NAME

vcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VCOSQF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VCOSQF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE COSQF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vcosqf(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vcosqf_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VCOSQF, a real two-dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave cosine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VCOSTI.

NAME

vcosqi - initialize the array WSAVE, which is used in both VCOSQF and VCOSQB.

SYNOPSIS

```
SUBROUTINE VCOSQI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE VCOSQI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VCOSQI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE VCOSQI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vcosqi(int n, float *wsave);

void vcosqi_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. VCOSQI needs to be called only once to initialize WSAVE before calling VCOSQF and/or VCOSQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vcost - compute the discrete Fourier cosine transform of an even sequence. The VCOST transform is normalized, so a call of VCOST followed by a call of VCOST will return the original sequence.

SYNOPSIS

```
SUBROUTINE VCOST( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VCOST_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COST( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE COST_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vcost(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vcost_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N - 1 is a product of small primes. N > = 2.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VCOST, a real two- dimensional array with dimensions of (MDIMX x (N+1)) whose rows contain the sequences to be transformed. On exit, the cosine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x (N-1)).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VCOSTI.

NAME

vcosti - initialize the array WSAVE, which is used in VCOST.

SYNOPSIS

```
SUBROUTINE VCOSTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE VCOSTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VCOSTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE VCOSTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vcosti(int n, float *wsave);

void vcosti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N - 1 is a product of small primes. N >
  = 2.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. VCOSTI is called once to initialize WSAVE before calling VCOST and need not be called again between calls to VCOST if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vdcosqb - synthesize a Fourier sequence from its representation in terms of a cosine series with odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDCOSQB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDCOSQB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE COSQB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdcosqb(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdcosqb_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, the rows contain the sequences to be transformed. On exit, the quarter-wave cosine synthesis of the input.
- **XT (input)**
  A work array.
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VCOSQI.

NAME

vdcosqf - compute the Fourier coefficients in a cosine series representation with only odd wave numbers. The VCOSQ operations are normalized, so a call of VCOSQF followed by a call of VCOSQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDCOSQF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDCOSQF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COSQF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE COSQF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdcosqf(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdcosqf_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VCOSQF, a real two-dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave cosine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VCOSQI.

NAME

vdcosqi - initialize the array WSAVE, which is used in both VCOSQF and VCOSQB.

SYNOPSIS

```
SUBROUTINE VDCOSQI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE VDCOSQI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VCOSQI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE VCOSQI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vdcosqi(int n, double *wsave);

void vdcosqi_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. VDCOSQI needs to be called only once to initialize WSAVE before calling VDCOSQF and/or VDCOSQB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vdcost - compute the discrete Fourier cosine transform of an even sequence. The VCOST transform is normalized, so a call of VCOST followed by a call of VCOST will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDCOST( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDCOST_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE COST( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE COST_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdcost(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdcost_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N - 1 is a product of small primes. N > = 2.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VCOST, a real two- dimensional array with dimensions of (MDIMX x (N+1)) whose rows contain the sequences to be transformed. On exit, the cosine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x (N-1)).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (2 * N + 15) or greater initialized by VDCOSTI.

NAME

vdcosti - initialize the array WSAVE, which is used in VCOST.

SYNOPSIS

```
SUBROUTINE VDCOSTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE VDCOSTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VCOSTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE VCOSTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vdcosti(int n, double *wsave);

void vdcosti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N - 1 is a product of small primes. N >
  = 2.
- **WSAVE (input/output)**
  On entry, an array of dimension (2 * N + 15) or greater. VDCOSTI is called once to initialize WSAVE before calling VDCOST and need not be called again between calls to VDCOST if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vdfftb - compute a periodic sequence from its Fourier coefficients. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDFFTB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDFFTB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE FFTB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdfftb(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdfftb_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VRFFTF, a real two-dimensional array `X(M,N)` whose rows contain the sequences to be transformed.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (N+15) or greater initialized by VRFFTI.

NAME

vdfftf - compute the Fourier coefficients of a periodic sequence. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDFFTF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDFFTF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE FFTF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdfftf(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdfftf_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VRFFTF, a real two- dimensional array `X(M,N)` whose rows contain the sequences to be transformed.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (N+15) or greater initialized by VRFFTI.

NAME

vdffti - initialize the array WSAVE, which is used in both VRFFTF and VRFFTB.

SYNOPSIS

```
SUBROUTINE VDFFTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE VDFFTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VFFTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE VFFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vdffti(int n, double *wsave);

void vdffti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (N + 15) or greater. VRFFTI needs to be called only once to initialize WSAVE before calling VRFFTF and/or VRFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vdsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDSINQB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDSINQB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE SINQB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdsinqb(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdsinqb_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, a real two- dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave sine synthesis of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least (2 * N + 15) for vector subroutines, initialized by VSINQI.

NAME

vdsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDSINQF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDSINQF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE SINQF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdsinqf(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdsinqf_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VSINQF, a real two-dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave sine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least (2 * N + 15), initialized by VSINQI.

NAME

vdsinqi - initialize the array WSAVE, which is used in both VSINQF and VSINQB.

SYNOPSIS

```
SUBROUTINE VDSINQI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE VDSINQI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VSINQI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE VSINQI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vdsinqi(int n, double *wsave);

void vdsinqi_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array with a dimension of at least (2 * N + 15). The same work array can be used for both VSINQF and VSINQB as long as N remains unchanged. Different WSAVE arrays are required for different values of N. This initialization does not have to be repeated between calls to VSINQF or VSINQB as long as N and WSAVE remain unchanged, thus subsequent transforms can be obtained faster than the first.

NAME

vdsint - compute the discrete Fourier sine transform of an odd sequence. The VSINT transforms are unnormalized inverses of themselves, so a call of VSINT followed by another call of VSINT will multiply the input sequence by 2 * (N+1). The VSINT transforms are normalized, so a call of VSINT followed by a call of VSINT will return the original sequence.

SYNOPSIS

```
SUBROUTINE VDSINT( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VDSINT_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
DOUBLE PRECISION X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINT( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT

SUBROUTINE SINT_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL(8), DIMENSION(:) :: WSAVE
REAL(8), DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vdsint(int m, int n, double *x, double *xt, int mdimx, double *wsave);

void vdsint_64(long m, long n, double *x, double *xt, long mdimx, double *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N+1 is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, a real two-dimensional array with dimensions of (MDIMX x (N+1)) whose rows contain the sequences to be transformed. On exit, the sine transform of the input.
- **XT (input/output)**
  A real two-dimensional work array with dimensions of (MDIMX x (N+1)).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least int(2.5 * N + 15) initialized by VSINTI.

NAME

vdsinti - initialize the array WSAVE, which is used in subroutine VSINT.

SYNOPSIS

```
SUBROUTINE VDSINTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE VDSINTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VSINTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE VSINTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vdsinti(int n, double *wsave);

void vdsinti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2N + N/2 + 15) or greater. VSINTI is called once to initialize WSAVE before calling VSINT and need not be called again between calls to VSINT if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vrfftb - compute a periodic sequence from its Fourier coefficients. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VRFFTB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VRFFTB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE FFTB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vrfftb(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vrfftb_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VRFFTF, a real two-dimensional array `X(M,N)` whose rows contain the sequences to be transformed.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input/output)**
  On entry, an array of dimension (N+15) or greater initialized by VRFFTI.

NAME

vrfftf - compute the Fourier coefficients of a periodic sequence. The VRFFT operations are normalized, so a call of VRFFTF followed by a call of VRFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VRFFTF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VRFFTF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE FFTF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vrfftf(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vrfftf_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VRFFTF, a real two- dimensional array `X(M,N)` whose rows contain the sequences to be transformed.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array of dimension (N+15) or greater initialized by VRFFTI.

NAME

vrffti - initialize the array WSAVE, which is used in both VRFFTF and VRFFTB.

SYNOPSIS

```
SUBROUTINE VRFFTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE VRFFTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VFFTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE VFFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vrffti(int n, float *wsave);

void vrffti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (N + 15) or greater. VRFFTI needs to be called only once to initialize WSAVE before calling VRFFTF and/or VRFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vsinqb - synthesize a Fourier sequence from its representation in terms of a sine series with odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VSINQB( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VSINQB_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQB( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE SINQB_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vsinqb(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vsinqb_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, a real two- dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave sine synthesis of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least (2 * N + 15) for vector subroutines, initialized by VSINQI.

NAME

vsinqf - compute the Fourier coefficients in a sine series representation with only odd wave numbers. The VSINQ operations are normalized, so a call of VSINQF followed by a call of VSINQB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VSINQF( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VSINQF_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINQF( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE SINQF_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vsinqf(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vsinqf_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed. For VSINQF, a real two-dimensional array with dimensions of (MDIMX x N) whose rows contain the sequences to be transformed. On exit, the quarter-wave sine transform of the input.
- **XT (input)**
  A real two-dimensional work array with dimensions of (MDIMX x N).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least (2 * N + 15), initialized by VSINQI.

NAME

vsinqi - initialize the array WSAVE, which is used in both VSINQF and VSINQB.

SYNOPSIS

```
SUBROUTINE VSINQI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE VSINQI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VSINQI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE VSINQI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vsinqi(int n, float *wsave);

void vsinqi_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. The method is most efficient when N is a product of small primes.
- **WSAVE (input/output)**
  On entry, an array with a dimension of at least (2 * N + 15). The same work array can be used for both VSINQF and VSINQB as long as N remains unchanged. Different WSAVE arrays are required for different values of N. This initialization does not have to be repeated between calls to VSINQF or VSINQB as long as N and WSAVE remain unchanged, thus subsequent transforms can be obtained faster than the first.

NAME

vsint - compute the discrete Fourier sine transform of an odd sequence. The VSINT transforms are unnormalized inverses of themselves, so a call of VSINT followed by another call of VSINT will multiply the input sequence by 2 * (N+1). The VSINT transforms are normalized, so a call of VSINT followed by a call of VSINT will return the original sequence.

SYNOPSIS

```
SUBROUTINE VSINT( M, N, X, XT, MDIMX, WSAVE)
INTEGER M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)

SUBROUTINE VSINT_64( M, N, X, XT, MDIMX, WSAVE)
INTEGER*8 M, N, MDIMX
REAL X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE SINT( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT

SUBROUTINE SINT_64( [M], [N], X, XT, [MDIMX], WSAVE)
INTEGER(8) :: M, N, MDIMX
REAL, DIMENSION(:) :: WSAVE
REAL, DIMENSION(:,:) :: X, XT
```

C INTERFACE

#include <sunperf.h>

void vsint(int m, int n, float *x, float *xt, int mdimx, float *wsave);

void vsint_64(long m, long n, float *x, float *xt, long mdimx, float *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N+1 is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, a real two-dimensional array with dimensions of (MDIMX x (N+1)) whose rows contain the sequences to be transformed. On exit, the sine transform of the input.
- **XT (input/output)**
  A real two-dimensional work array with dimensions of (MDIMX x (N+1)).
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **WSAVE (input)**
  On entry, an array with dimension of at least int(2.5 * N + 15) initialized by VSINTI.

NAME

vsinti - initialize the array WSAVE, which is used in subroutine VSINT.

SYNOPSIS

```
SUBROUTINE VSINTI( N, WSAVE)
INTEGER N
REAL WSAVE(*)

SUBROUTINE VSINTI_64( N, WSAVE)
INTEGER*8 N
REAL WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE VSINTI( N, WSAVE)
INTEGER :: N
REAL, DIMENSION(:) :: WSAVE

SUBROUTINE VSINTI_64( N, WSAVE)
INTEGER(8) :: N
REAL, DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void vsinti(int n, float *wsave);

void vsinti_64(long n, float *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (2N + N/2 + 15) or greater. VSINTI is called once to initialize WSAVE before calling VSINT and need not be called again between calls to VSINT if N and WSAVE remain unchanged. Thus, subsequent transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

vzfftb - compute a periodic sequence from its Fourier coefficients. The VZFFT operations are normalized, so a call of VZFFTF followed by a call of VZFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VZFFTB( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
DOUBLE COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER M, N, MDIMX

SUBROUTINE VZFFTB_64( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
DOUBLE COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER*8 M, N, MDIMX
```

F95 INTERFACE

```
SUBROUTINE FFTB( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX(8), DIMENSION(:) :: WSAVE
COMPLEX(8), DIMENSION(:,:) :: X, XT
INTEGER :: M, N, MDIMX

SUBROUTINE FFTB_64( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX(8), DIMENSION(:) :: WSAVE
COMPLEX(8), DIMENSION(:,:) :: X, XT
INTEGER(8) :: M, N, MDIMX
```

C INTERFACE

#include <sunperf.h>

void vzfftb(int m, int n, doublecomplex *x, doublecomplex *xt, int mdimx, char rowcol, doublecomplex *wsave);

void vzfftb_64(long m, long n, doublecomplex *x, doublecomplex *xt, long mdimx, char rowcol, doublecomplex *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M > = 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, the rows contain the sequences to be transformed.
- **XT (input)**
  A work array.
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX > = M.
- **ROWCOL (input)**
  Indicates whether to transform rows ('R' or 'r') or columns ('C' or 'c').
- **WSAVE (input/output)**
  On entry, an array of dimension (K+15) or greater, where K = M if ROWCOL = ('R' or 'r'). Otherwise, K = N. WSAVE is initialized by VZFFTI.

NAME

vzfftf - compute the Fourier coefficients of a periodic sequence. The VZFFT operations are normalized, so a call of VZFFTF followed by a call of VZFFTB will return the original sequence.

SYNOPSIS

```
SUBROUTINE VZFFTF( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
DOUBLE COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER M, N, MDIMX

SUBROUTINE VZFFTF_64( M, N, X, XT, MDIMX, ROWCOL, WSAVE)
CHARACTER * 1 ROWCOL
DOUBLE COMPLEX X(MDIMX,*), XT(MDIMX,*), WSAVE(*)
INTEGER*8 M, N, MDIMX
```

F95 INTERFACE

```
SUBROUTINE FFTF( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX(8), DIMENSION(:) :: WSAVE
COMPLEX(8), DIMENSION(:,:) :: X, XT
INTEGER :: M, N, MDIMX

SUBROUTINE FFTF_64( [M], [N], X, XT, [MDIMX], ROWCOL, WSAVE)
CHARACTER(LEN=1) :: ROWCOL
COMPLEX(8), DIMENSION(:) :: WSAVE
COMPLEX(8), DIMENSION(:,:) :: X, XT
INTEGER(8) :: M, N, MDIMX
```

C INTERFACE

#include <sunperf.h>

void vzfftf(int m, int n, doublecomplex *x, doublecomplex *xt, int mdimx, char rowcol, doublecomplex *wsave);

void vzfftf_64(long m, long n, doublecomplex *x, doublecomplex *xt, long mdimx, char rowcol, doublecomplex *wsave);

ARGUMENTS

- **M (input)**
  The number of sequences to be transformed. M >= 0.
- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **X (input/output)**
  On entry, an array X(M,N) whose rows contain the sequences to be transformed.
- **XT (input)**
  A work array.
- **MDIMX (input)**
  Leading dimension of the arrays X and XT as specified in a dimension or type statement. MDIMX >= M.
- **ROWCOL (input)**
  Indicates whether to transform rows ('R' or 'r') or columns ('C' or 'c').
- **WSAVE (input)**
  On entry, an array of dimension (K+15) or greater, where K = M if ROWCOL = ('R' or 'r'). Otherwise, K = N. WSAVE is initialized by VZFFTI.

NAME

vzffti - initialize the array WSAVE, which is used in both VZFFTF and VZFFTB.

SYNOPSIS

```
SUBROUTINE VZFFTI( N, WSAVE)
DOUBLE COMPLEX WSAVE(*)
INTEGER N

SUBROUTINE VZFFTI_64( N, WSAVE)
DOUBLE COMPLEX WSAVE(*)
INTEGER*8 N
```

F95 INTERFACE

```
SUBROUTINE VFFTI( N, WSAVE)
COMPLEX(8), DIMENSION(:) :: WSAVE
INTEGER :: N

SUBROUTINE VFFTI_64( N, WSAVE)
COMPLEX(8), DIMENSION(:) :: WSAVE
INTEGER(8) :: N
```

C INTERFACE

#include <sunperf.h>

void vzffti(int n, doublecomplex *wsave);

void vzffti_64(long n, doublecomplex *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (N + 15) or greater. VZFFTI needs to be called only once to initialize WSAVE before calling VZFFTF and/or VZFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

zaxpy - compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE ZAXPY( N, ALPHA, X, INCX, Y, INCY)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE ZAXPY_64( N, ALPHA, X, INCX, Y, INCY)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE AXPY( [N], ALPHA, X, [INCX], Y, [INCY])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE AXPY_64( [N], ALPHA, X, [INCX], Y, [INCY])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zaxpy(int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy);

void zaxpy_64(long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy);

PURPOSE

zaxpy compute y := alpha * x + y where alpha is a scalar and x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  array of DIMENSION at least (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  array of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zaxpyi - Compute y := alpha * x + y

SYNOPSIS

```
SUBROUTINE ZAXPYI(NZ, A, X, INDX, Y)

DOUBLE COMPLEX A
DOUBLE COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE ZAXPYI_64(NZ, A, X, INDX, Y)

DOUBLE COMPLEX A
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE AXPYI([NZ], [A], X, INDX, Y)

```
COMPLEX(8) :: A
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE AXPYI_64([NZ], [A], X, INDX, Y)

COMPLEX(8) :: A
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZAXPYI Compute y := alpha * x + y where alpha is a scalar, x is a sparse vector, and y is a vector in full storage form

```
do i = 1, n
  y(indx(i)) = alpha * x(i) + y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

A (input)

On entry, ALPHA specifies the scaling value. Unchanged on exit.

X (input)

Vector containing the values of the compressed form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector on input which contains the vector Y in full storage form. On exit, only the elements corresponding to the indices in INDX have been modified.

NAME

zbdsqr - compute the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B.

SYNOPSIS

```
 SUBROUTINE ZBDSQR( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U, LDU,
*      C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX VT(LDVT,*), U(LDU,*), C(LDC,*)
 INTEGER N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 DOUBLE PRECISION D(*), E(*), WORK(*)

 SUBROUTINE ZBDSQR_64( UPLO, N, NCVT, NRU, NCC, D, E, VT, LDVT, U,
*      LDU, C, LDC, WORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX VT(LDVT,*), U(LDU,*), C(LDC,*)
 INTEGER*8 N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 DOUBLE PRECISION D(*), E(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE BDSQR( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT, [LDVT],
*       U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:,:) :: VT, U, C
 INTEGER :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK

 SUBROUTINE BDSQR_64( UPLO, [N], [NCVT], [NRU], [NCC], D, E, VT,
*       [LDVT], U, [LDU], C, [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:,:) :: VT, U, C
 INTEGER(8) :: N, NCVT, NRU, NCC, LDVT, LDU, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void zbdsqr(char uplo, int n, int ncvt, int nru, int ncc, double *d, double *e, doublecomplex *vt, int ldvt, doublecomplex *u, int ldu, doublecomplex *c, int ldc, int *info);

void zbdsqr_64(char uplo, long n, long ncvt, long nru, long ncc, double *d, double *e, doublecomplex *vt, long ldvt, doublecomplex *u, long ldu, doublecomplex *c, long ldc, long *info);

PURPOSE

zbdsqr computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = Q * S * P' (P' denotes the transpose of P), where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and Q and P are orthogonal matrices.

The routine computes S, and optionally computes U * Q, P' * VT, or Q' * C, for given complex input matrices U, VT, and C.

See ``Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy," by J. Demmel and W. Kahan, LAPACK Working Note #3 (or SIAM J. Sci. Statist. Comput. vol. 11, no. 5, pp. 873-912, Sept 1990) and

``Accurate singular values and differential qd algorithms," by B. Parlett and V. Fernando, Technical Report CPAM-554, Mathematics Department, University of California at Berkeley, July 1992 for a detailed description of the algorithm.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  B is upper bidiagonal;

  = 'L':  B is lower bidiagonal.
  ```

- **N (input)**
  The order of the matrix B. N > = 0.
- **NCVT (input)**
  The number of columns of the matrix VT. NCVT > = 0.
- **NRU (input)**
  The number of rows of the matrix U. NRU > = 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the bidiagonal matrix B. On exit, if INFO =0, the singular values of B in decreasing order.
- **E (input/output)**
  On entry, the elements of E contain the offdiagonal elements of of the bidiagonal matrix whose SVD is desired. On normal exit (INFO = 0), E is destroyed. If the algorithm does not converge (INFO > 0), D and E will contain the diagonal and superdiagonal elements of a bidiagonal matrix orthogonally equivalent to the one given as input. E(N) is used for workspace.
- **VT (input/output)**

On entry, an N-by-NCVT matrix VT. On exit, VT is overwritten by P' * VT. VT is not referenced if NCVT = 0.

- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= `max(1,N)` if NCVT > 0; LDVT >= 1 if NCVT = 0.
- **U (input/output)**
  On entry, an NRU-by-N matrix U. On exit, U is overwritten by U * Q. U is not referenced if NRU = 0.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,NRU).
- **C (input/output)**
  On entry, an N-by-NCC matrix C. On exit, C is overwritten by Q' * C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,N)` if NCC > 0; LDC >=1 if NCC = 0.
- **WORK (workspace)**
  dimension (4*N)
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  If INFO  = -i, the i-th argument had an illegal value

 > 0:  the algorithm did not converge; D and E contain the
elements of a bidiagonal matrix which is orthogonally
similar to the input matrix B;  if INFO  = i, i
elements of E have not converged to zero.
```

NAME

zcnvcor - compute the convolution or correlation of complex vectors

---

SYNOPSIS

```
 SUBROUTINE ZCNVCOR( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M, Y,
*      IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 DOUBLE COMPLEX X(*), Y(*), Z(*), WORK(*)
 INTEGER NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK

 SUBROUTINE ZCNVCOR_64( CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M,
*      Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK)
 CHARACTER * 1 CNVCOR, FOUR
 DOUBLE COMPLEX X(*), Y(*), Z(*), WORK(*)
 INTEGER*8 NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE, M,
*       Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 COMPLEX(8), DIMENSION(:) :: X, Y, Z, WORK
 INTEGER :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK

 SUBROUTINE CNVCOR_64( CNVCOR, FOUR, [NX], X, IFX, [INCX], NY, NPRE,
*       M, Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK,
*       [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, FOUR
 COMPLEX(8), DIMENSION(:) :: X, Y, Z, WORK
 INTEGER(8) :: NX, IFX, INCX, NY, NPRE, M, IFY, INC1Y, INC2Y, NZ, K, IFZ, INC1Z, INC2Z, LWORK
```

C INTERFACE

#include <sunperf.h>

void zcnvcor(char cnvcor, char four, int nx, doublecomplex *x, int ifx, int incx, int ny, int npre, int m, doublecomplex *y, int ify, int inc1y, int inc2y, int nz, int k, doublecomplex *z, int ifz, int inc1z, int inc2z, doublecomplex *work, int lwork);

void zcnvcor_64(char cnvcor, char four, long nx, doublecomplex *x, long ifx, long incx, long ny, long npre, long m, doublecomplex *y, long ify, long inc1y, long inc2y, long nz, long k, doublecomplex *z, long ifz, long inc1z, long inc2z, doublecomplex *work, long lwork);

PURPOSE

zcnvcor computes the convolution or correlation of complex vectors.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' if convolution is desired, 'R' or 'r' if correlation is desired.
- **FOUR (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' if the computation should be done directly from the definition. The Fourier transform method is generally faster, but it may introduce noticeable errors into certain results, notably when both the real and imaginary parts of the filter and data vectors consist entirely of integers or vectors where elements of either the filter vector or a given data vector differ significantly in magnitude from the 1-norm of the vector.
- **NX (input)**
  Length of the filter vector. NX > = 0. ZCNVCOR will return immediately if NX = 0.
- **X (input)**

  ```
  dimension(*)
  ```

  Filter vector.
- **IFX (input)**
  Index of the first element of X. NX > = IFX > = 1.
- **INCX (input)**
  Stride between elements of the filter vector in X. INCX > 0.
- **NY (input)**
  Length of the input vectors. NY > = 0. ZCNVCOR will return immediately if NY = 0.
- **NPRE (input)**
  The number of implicit zeros prepended to the Y vectors. NPRE > = 0.
- **M (input)**
  Number of input vectors. M > = 0. ZCNVCOR will return immediately if M = 0.
- **Y (input)**

  ```
  dimension(*)
  ```

  Input vectors.
- **IFY (input)**
  Index of the first element of Y. NY > = IFY > = 1.
- **INC1Y (input)**
  Stride between elements of the input vectors in Y. INC1Y > 0.
- **INC2Y (input)**
  Stride between the input vectors in Y. INC2Y > 0.
- **NZ (input)**
  Length of the output vectors. NZ > = 0. ZCNVCOR will return immediately if NZ = 0. See the Notes section below for information about how this argument interacts with NX and NY to control circular versus end-off shifting.
- **K (input)**
  Number of Z vectors. K > = 0. If K = 0 then ZCNVCOR will return immediately. If K < M then only the first K input vectors will be processed. If K > M then M input vectors will be processed.
- **Z (output)**

  ```
  dimension(*)
  ```

  Result vectors.
- **IFZ (input)**

Index of the first element of Z. NZ > = IFZ > = 1.

- **INC1Z (input)**
  Stride between elements of the output vectors in Z. INC1Z > 0.
- **INC2Z (input)**
  Stride between the output vectors in Z. INC2Z > 0.
- **WORK (input/output)**
  (input/scratch) `dimension(LWORK)`

  Scratch space. Before the first call to ZCNVCOR with particular values of the integer arguments the first element of WORK must be set to zero. If WORK is written between calls to ZCNVCOR or if ZCNVCOR is called with different values of the integer arguments then the first element of WORK must again be set to zero before each call. If WORK has not been written and the same values of the integer arguments are used then the first element of WORK to zero. This can avoid certain initializations that store their results into WORK, and avoiding the initialization can make ZCNVCOR run faster.
- **LWORK (input)**
  Length of WORK. LWORK > = 2*MAX(NX,NY,NZ)+8.

NAME

zcnvcor2 - compute the convolution or correlation of complex matrices

---

SYNOPSIS

```
 SUBROUTINE ZCNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*       LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*), Z(LDZ,*), WORKIN(*)
 INTEGER MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK

 SUBROUTINE ZCNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, MX, NX, X, LDX, MY, NY, MPRE, NPRE, Y, LDY, MZ, NZ, Z,
*       LDZ, WORKIN, LWORK)
 CHARACTER * 1 CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*), Z(LDZ,*), WORKIN(*)
 INTEGER*8 MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
```

F95 INTERFACE

```
 SUBROUTINE CNVCOR2( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX(8), DIMENSION(:) :: WORKIN
 COMPLEX(8), DIMENSION(:,:) :: X, Y, Z
 INTEGER :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK

 SUBROUTINE CNVCOR2_64( CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY,
*       SCRATCHY, [MX], [NX], X, [LDX], [MY], [NY], MPRE, NPRE, Y, [LDY],
*       [MZ], [NZ], Z, [LDZ], WORKIN, [LWORK])
 CHARACTER(LEN=1) :: CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY
 COMPLEX(8), DIMENSION(:) :: WORKIN
 COMPLEX(8), DIMENSION(:,:) :: X, Y, Z
 INTEGER(8) :: MX, NX, LDX, MY, NY, MPRE, NPRE, LDY, MZ, NZ, LDZ, LWORK
```

C INTERFACE

#include <sunperf.h>

void zcnvcor2(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, int mx, int nx, doublecomplex *x, int ldx, int my, int ny, int mpre, int npre, doublecomplex *y, int ldy, int mz, int nz, doublecomplex *z, int ldz, doublecomplex *workin, int lwork);

void zcnvcor2_64(char cnvcor, char method, char transx, char scratchx, char transy, char scratchy, long mx, long nx, doublecomplex *x, long ldx, long my, long ny, long mpre, long npre, doublecomplex *y, long ldy, long mz, long nz, doublecomplex *z, long ldz, doublecomplex *workin, long lwork);

PURPOSE

zcnvcor2 computes the convolution or correlation of complex matrices.

ARGUMENTS

- **CNVCOR (input)**
  \'V' or 'v' to compute convolution, 'R' or 'r' to compute correlation.
- **METHOD (input)**
  \'T' or 't' if the Fourier transform method is to be used, 'D' or 'd' to compute directly from the definition.
- **TRANSX (input)**
  \'N' or 'n' if X is the filter matrix, 'T' or 't' if `transpose(X)` is the filter matrix.
- **SCRATCHX (input)**
  \'N' or 'n' if X must be preserved, 'S' or 's' if X can be used as scratch space. The contents of X are undefined after returning from a call in which X is allowed to be used for scratch.
- **TRANSY (input)**
  \'N' or 'n' if Y is the input matrix, 'T' or 't' if `transpose(Y)` is the input matrix.
- **SCRATCHY (input)**
  \'N' or 'n' if Y must be preserved, 'S' or 's' if Y can be used as scratch space. The contents of Y are undefined after returning from a call in which Y is allowed to be used for scratch.
- **MX (input)**
  Number of rows in the filter matrix. MX > = 0.
- **NX (input)**
  Number of columns in the filter matrix. NX > = 0.
- **X (input)**

  ```
  dimension(LDX,NX)
  ```

  On entry, the filter matrix. Unchanged on exit if SCRATCHX is 'N' or 'n', undefined on exit if SCRATCHX is 'S' or 's'.
- **LDX (input)**
  Leading dimension of the array that contains the filter matrix.
- **MY (input)**
  Number of rows in the input matrix. MY > = 0.

- **NY (input)**
  Number of columns in the input matrix. NY > = 0.
- **MPRE (input)**
  Number of implicit zeros to prepend to each row of the input matrix. MPRE > = 0.
- **NPRE (input)**
  Number of implicit zeros to prepend to each column of the input matrix. NPRE > = 0.
- **Y (input)**

  ```
  dimension(LDY,*)
  ```

  Input matrix. Unchanged on exit if SCRATCHY is 'N' or 'n', undefined on exit if SCRATCHY is 'S' or 's'.
- **LDY (input)**
  Leading dimension of the array that contains the input matrix.
- **MZ (input)**
  Number of rows in the output matrix. MZ > = 0. ZCNVCOR2 will return immediately if MZ = 0.
- **NZ (input)**
  Number of columns in the output matrix. NZ > = 0. ZCNVCOR2 will return immediately if NZ = 0.
- **Z (output)**

  ```
  dimension(LDZ,*)
  ```

  Result matrix.
- **LDZ (input)**
  Leading dimension of the array that contains the result matrix. LDZ > = MAX(1,MZ).
- **WORKIN (input/output)**
  (input/scratch) `dimension(LWORK)`

  On entry for the first call to ZCNVCOR2, `WORKIN(1)` must contain CMPLX(0.0,0.0). After the first call, `WORKIN(1)` must be set to `CMPLX(0.0,0.0)` iff WORKIN has been altered since the last call to this subroutine or if the sizes of the arrays have changed.
- **LWORK (input)**
  Length of the work vector. If the FFT is to be used then for best performance LWORK should be at least 30 words longer than the amount of memory needed to hold the trig tables. If the FFT is not used, the value of LWORK is unimportant.

NAME

zcopy - Copy x to y

SYNOPSIS

```
SUBROUTINE ZCOPY( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE ZCOPY_64( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE COPY( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE COPY_64( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zcopy(int n, doublecomplex *x, int incx, doublecomplex *y, int incy);

void zcopy_64(long n, doublecomplex *x, long incx, doublecomplex *y, long incy);

PURPOSE

zcopy Copy x to y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  of DIMENSION at least (1 + (m - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zdotc - compute the dot product of two vectors `conjg(x)` and y.

SYNOPSIS

```
DOUBLE COMPLEX FUNCTION ZDOTC( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

DOUBLE COMPLEX FUNCTION ZDOTC_64( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
COMPLEX(8) FUNCTION DOTC( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

COMPLEX(8) FUNCTION DOTC_64( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

doublecomplex zdotc(int n, doublecomplex *x, int incx, doublecomplex *y, int incy);

doublecomplex zdotc_64(long n, doublecomplex *x, long incx, doublecomplex *y, long incy);

PURPOSE

zdotc compute the dot product of `conjg(x)` and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zdotci - Compute the complex conjugated indexed dot product.

SYNOPSIS

```
DOUBLE COMPLEX FUNCTION ZDOTCI(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

DOUBLE COMPLEX FUNCTION ZDOTCI_64(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE DOUBLE COMPLEX FUNCTION DOTCI([NZ], X, INDX, Y)

```
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

DOUBLE COMPLEX FUNCTION DOTCI_64([NZ], X, INDX, Y)

COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZDOTCI Compute the complex conjugated indexed dot product of a complex sparse vector x stored in compressed form with a complex vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + conjg(x(i)) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

zdotu - compute the dot product of two vectors x and y.

SYNOPSIS

```
DOUBLE COMPLEX FUNCTION ZDOTU( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

DOUBLE COMPLEX FUNCTION ZDOTU_64( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
COMPLEX(8) FUNCTION DOT( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

COMPLEX(8) FUNCTION DOT_64( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

doublecomplex zdotu(int n, doublecomplex *x, int incx, doublecomplex *y, int incy);

doublecomplex zdotu_64(long n, doublecomplex *x, long incx, doublecomplex *y, long incy);

PURPOSE

zdotu compute the dot product of x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. If N is not positive then the function returns the value 0.0. Unchanged on exit.
- **X (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  of DIMENSION at least (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zdotui - Compute the complex unconjugated indexed dot product.

SYNOPSIS

```
DOUBLE COMPLEX FUNCTION CDOTCI(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

DOUBLE COMPLEX FUNCTION CDOTCI_64(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE DOUBLE COMPLEX FUNCTION DOTCI([NZ], X, INDX, Y)

```
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

DOUBLE COMPLEX FUNCTION DOTCI_64([NZ], X, INDX, Y)

COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZDOTUI Compute the complex unconjugated indexed dot product of a complex sparse vector x stored in compressed form with a complex vector y in full storage form.

```
dot = 0
do i = 1, n
  dot = dot + x(i) * y(indx(i))
enddo
```

ARGUMENTS

NZ (input)

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector in compressed form. Unchanged on exit.

INDX (input)

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (input)

Vector in full storage form. Only the elements corresponding to the indices in INDX will be accessed.

NAME

zdrot - Apply a plane rotation.

SYNOPSIS

```
SUBROUTINE ZDROT( N, CX, INCX, CY, INCY, C, S)
DOUBLE COMPLEX CX(*), CY(*)
INTEGER N, INCX, INCY
DOUBLE PRECISION C, S

SUBROUTINE ZDROT_64( N, CX, INCX, CY, INCY, C, S)
DOUBLE COMPLEX CX(*), CY(*)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION C, S
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], CX, [INCX], CY, [INCY], C, S)
COMPLEX(8), DIMENSION(:) :: CX, CY
INTEGER :: N, INCX, INCY
REAL(8) :: C, S

SUBROUTINE ROT_64( [N], CX, [INCX], CY, [INCY], C, S)
COMPLEX(8), DIMENSION(:) :: CX, CY
INTEGER(8) :: N, INCX, INCY
REAL(8) :: C, S
```

C INTERFACE

#include <sunperf.h>

void zdrot(int n, doublecomplex *cx, int incx, doublecomplex *cy, int incy, double c, double s);

void zdrot_64(long n, doublecomplex *cx, long incx, doublecomplex *cy, long incy, double c, double s);

PURPOSE

zdrot Apply a plane rotation, where the cos and sin (c and s) are real and the vectors x and y are complex.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **CX (input)**
  Before entry, the incremented array CX must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of CX. INCX must not be zero. Unchanged on exit.
- **CY (output)**
  On entry, the incremented array CY must contain the vector y. On exit, CY is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of CY. INCY must not be zero. Unchanged on exit.
- **C (input)**
  On entry, the cosine. Unchanged on exit.
- **S (input)**
  On entry, the sin. Unchanged on exit.

NAME

zdscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE ZDSCAL( N, ALPHA, Y, INCY)
DOUBLE COMPLEX Y(*)
INTEGER N, INCY
DOUBLE PRECISION ALPHA

SUBROUTINE ZDSCAL_64( N, ALPHA, Y, INCY)
DOUBLE COMPLEX Y(*)
INTEGER*8 N, INCY
DOUBLE PRECISION ALPHA
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
COMPLEX(8), DIMENSION(:) :: Y
INTEGER :: N, INCY
REAL(8) :: ALPHA

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
COMPLEX(8), DIMENSION(:) :: Y
INTEGER(8) :: N, INCY
REAL(8) :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void zdscal(int n, double alpha, doublecomplex *y, int incy);

void zdscal_64(long n, double alpha, doublecomplex *y, long incy);

PURPOSE

zdscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zfft2b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFT2F followed by a call of ZFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
SUBROUTINE ZFFT2B( M, N, A, LDA, WORK, LWORK)
DOUBLE COMPLEX A(LDA,*)
INTEGER M, N, LDA, LWORK
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT2B_64( M, N, A, LDA, WORK, LWORK)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, LWORK
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2B( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT2B_64( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft2b(int m, int n, doublecomplex *a, int lda, double *work, int lwork);

void zfft2b_64(long m, long n, doublecomplex *a, long lda, double *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M > = 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **A (input/output)**
  On entry, a two-dimensional array `A(M,N)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA > = M.
- **WORK (input)**
  On entry, an array with dimension of at least LWORK. WORK must have been initialized by ZFFT2I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = (4 * (M + N) + 30)

NAME

zfft2f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFT2F followed by a call of ZFFT2B will multiply the input sequence by M*N.

SYNOPSIS

```
SUBROUTINE ZFFT2F( M, N, A, LDA, WORK, LWORK)
DOUBLE COMPLEX A(LDA,*)
INTEGER M, N, LDA, LWORK
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT2F_64( M, N, A, LDA, WORK, LWORK)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, LWORK
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT2F( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT2F_64( [M], [N], A, [LDA], WORK, LWORK)
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft2f(int m, int n, doublecomplex *a, int lda, double *work, int lwork);

void zfft2f_64(long m, long n, doublecomplex *a, long lda, double *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M > = 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **A (input/output)**
  On entry, a two-dimensional array `A(M,N)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA > = M.
- **WORK (input)**
  On input, workspace WORK must have been initialized by ZFFT2I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = (4 * (M + N) + 30)

NAME

zfft2i - initialize the array WSAVE, which is used in both the forward and backward transforms.

SYNOPSIS

```
SUBROUTINE ZFFT2I( M, N, WORK)
INTEGER M, N
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT2I_64( M, N, WORK)
INTEGER*8 M, N
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ZFFT2I( M, N, WORK)
INTEGER :: M, N
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE ZFFT2I_64( M, N, WORK)
INTEGER(8) :: M, N
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft2i(int m, int n, double *work);

void zfft2i_64(long m, long n, double *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (4 * (M + N) + 30) or greater. ZFFT2I needs to be called only once to initialize array WORK before calling ZFFT2F and/or ZFFT2B if M, N and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

zfft3b - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFT3F followed by a call of ZFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
SUBROUTINE ZFFT3B( M, N, K, A, LDA, LD2A, WORK, LWORK)
DOUBLE COMPLEX A(LDA,LD2A,*)
INTEGER M, N, K, LDA, LD2A, LWORK
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT3B_64( M, N, K, A, LDA, LD2A, WORK, LWORK)
DOUBLE COMPLEX A(LDA,LD2A,*)
INTEGER*8 M, N, K, LDA, LD2A, LWORK
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3B( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX(8), DIMENSION(:,:,:) :: A
INTEGER :: M, N, K, LDA, LD2A, LWORK
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT3B_64( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX(8), DIMENSION(:,:,:) :: A
INTEGER(8) :: M, N, K, LDA, LD2A, LWORK
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft3b(int m, int n, int k, doublecomplex *a, int lda, int ld2a, double *work, int lwork);

void zfft3b_64(long m, long n, long k, doublecomplex *a, long lda, long ld2a, double *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **K (input)**
  Number of planes to be transformed. These subroutines are most efficient when K is a product of small primes. K >= 0.
- **A (input/output)**
  On entry, a three-dimensional array `A(LDA,LD2A,K)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **LD2A (input)**
  Second dimension of the array containing the data to be transformed. LD2A >= N.
- **WORK (input)**
  On input, workspace WORK must have been initialized by ZFFT3I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4*(M + N + K) + 45).

NAME

zfft3f - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFT3F followed by a call of ZFFT3B will multiply the input sequence by M*N*K.

SYNOPSIS

```
SUBROUTINE ZFFT3F( M, N, K, A, LDA, LD2A, WORK, LWORK)
DOUBLE COMPLEX A(LDA,LD2A,*)
INTEGER M, N, K, LDA, LD2A, LWORK
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT3F_64( M, N, K, A, LDA, LD2A, WORK, LWORK)
DOUBLE COMPLEX A(LDA,LD2A,*)
INTEGER*8 M, N, K, LDA, LD2A, LWORK
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE FFT3F( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX(8), DIMENSION(:,:,:) :: A
INTEGER :: M, N, K, LDA, LD2A, LWORK
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE FFT3F_64( [M], [N], [K], A, [LDA], LD2A, WORK, LWORK)
COMPLEX(8), DIMENSION(:,:,:) :: A
INTEGER(8) :: M, N, K, LDA, LD2A, LWORK
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft3f(int m, int n, int k, doublecomplex *a, int lda, int ld2a, double *work, int lwork);

void zfft3f_64(long m, long n, long k, doublecomplex *a, long lda, long ld2a, double *work, long lwork);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. These subroutines are most efficient when M is a product of small primes. M >= 0.
- **N (input)**
  Number of columns to be transformed. These subroutines are most efficient when N is a product of small primes. N >= 0.
- **K (input)**
  Number of planes to be transformed. These subroutines are most efficient when K is a product of small primes. K >= 0.
- **A (input/output)**
  On entry, a three-dimensional array `A(M,N,K)` that contains the sequences to be transformed.
- **LDA (input)**
  Leading dimension of the array containing the data to be transformed. LDA >= M.
- **LD2A (input)**
  Second dimension of the array containing the data to be transformed. LD2A >= N.
- **WORK (input)**
  On input, workspace WORK must have been initialized by ZFFT3I.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= (4*(M + N + K) + 45).

NAME

zfft3i - initialize the array WSAVE, which is used in both ZFFT3F and ZFFT3B.

SYNOPSIS

```
SUBROUTINE ZFFT3I( M, N, K, WORK)
INTEGER M, N, K
DOUBLE PRECISION WORK(*)

SUBROUTINE ZFFT3I_64( M, N, K, WORK)
INTEGER*8 M, N, K
DOUBLE PRECISION WORK(*)
```

F95 INTERFACE

```
SUBROUTINE ZFFT3I( M, N, K, WORK)
INTEGER :: M, N, K
REAL(8), DIMENSION(:) :: WORK

SUBROUTINE ZFFT3I_64( M, N, K, WORK)
INTEGER(8) :: M, N, K
REAL(8), DIMENSION(:) :: WORK
```

C INTERFACE

#include <sunperf.h>

void zfft3i(int m, int n, int k, double *work);

void zfft3i_64(long m, long n, long k, double *work);

ARGUMENTS

- **M (input)**
  Number of rows to be transformed. M > = 0.
- **N (input)**
  Number of columns to be transformed. N > = 0.
- **K (input)**
  Number of planes to be transformed. K > = 0.
- **WORK (input/output)**
  On entry, an array of dimension (4*(M + N + K) + 45) or greater. ZFFT3I needs to be called only once to initialize array WORK before calling ZFFT3F and/or ZFFT3B if M, N, K and WORK remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

zfftb - compute a periodic sequence from its Fourier coefficients. The FFT operations are unnormalized, so a call of ZFFTF followed by a call of ZFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE ZFFTB( N, X, WSAVE)
DOUBLE COMPLEX X(*)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE ZFFTB_64( N, X, WSAVE)
DOUBLE COMPLEX X(*)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTB( [N], X, WSAVE)
COMPLEX(8), DIMENSION(:) :: X
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE FFTB_64( [N], X, WSAVE)
COMPLEX(8), DIMENSION(:) :: X
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void zfftb(int n, doublecomplex *x, double *wsave);

void zfftb_64(long n, doublecomplex *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (4 * N + 15) or greater and must have been initialized by ZFFTI.

NAME

zfftd - initialize the trigonometric weight and factor tables or compute the inverse Fast Fourier Transform of a double complex sequence. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTD( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX X(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(*), TRIGS(*), WORK(*)

 SUBROUTINE ZFFTD_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX X(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, N, [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX(8), DIMENSION(:) :: X
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: Y, TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, N, [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX(8), DIMENSION(:) :: X
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: Y, TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void zfftd(int iopt, int n, double scale, doublecomplex *x, double *y, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftd_64(long iopt, long n, double scale, doublecomplex *x, double *y, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftd initializes the trigonometric weight and factor tables or computes the inverse Fast Fourier Transform of a double complex sequence as follows: .Ve

```
                N-1
```

Y(k) = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

k ranges from 0 to N-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

In complex-to-real transform of length N, the (N/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored. Furthermore, due to symmetries the imaginary of the component of X(0) and X(N/2) (if N is even in the latter) is assumed to be zero and is not referenced.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  On entry, X is a double complex array whose first (N/2+1) elements are the input sequence to be transformed.
- **Y (output)**
  Double precision array of dimension at least N that contains the transform results. X and Y may be the same array starting at the same memory location. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **TRIGS (input/output)**
  Double precision array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

zfftd2 - initialize the trigonometric weight and factor tables or compute the two-dimensional inverse Fast Fourier Transform of a two-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTD2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY,*), TRIGS(*), WORK(*)

 SUBROUTINE ZFFTD2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: Y

 SUBROUTINE FFT2_64( IOPT, N1, [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void zfftd2(int iopt, int n1, int n2, double scale, doublecomplex *x, int ldx, double *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftd2_64(long iopt, long n1, long n2, double scale, doublecomplex *x, long ldx, double *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftd2 initializes the trigonometric weight and factor tables or computes the two-dimensional inverse Fast Fourier Transform of a two-dimensional double complex array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the rows of the input array. One-dimensional FFTs are then computed along the columns of the intermediate results.
.Ve

```
                    N1-1  N2-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j1=0  j2=0
```

.Ve

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

In complex-to-real transform of length N1, the (N1/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small

primes. N1 > = 0. Unchanged on exit.

- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX, N2) that contains input data to be transformed.
- **LDX (input)**
  Leading dimension of X. LDX > = (N1/2 + 1) Unchanged on exit.
- **Y (output)**
  Y is a double precision array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = 2*LDX Else LDY > = 2*LDX and LDY must be even. Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least `MAX(N1,2*N2)` where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1/2+1)

  -5 = LDY not equal 2*LDX when X and Y are same array

  -6 = (LDY < 2*LDX or LDY odd) when X and Y are same array

  -7 = (LWORK not equal 0) and (LWORK < `MAX(N1,2*N2))`

  -8 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output subarray Y(1:LDY, 1:N2) is overwritten.

NAME

zfftd3 - initialize the trigonometric weight and factor tables or compute the three-dimensional inverse Fast Fourier Transform of a three-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTD3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX1,LDX2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY1,LDY2,*), TRIGS(*), WORK(*)

 SUBROUTINE ZFFTD3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX1,LDX2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY1,LDY2,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, N1, [N2], [N3], [SCALE], X, [LDX1], LDX2, Y,
*       [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: X
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:,:) :: Y

 SUBROUTINE FFT3_64( IOPT, N1, [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: X
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void zfftd3(int iopt, int n1, int n2, int n3, double scale, doublecomplex *x, int ldx1, int ldx2, double *y, int ldy1, int ldy2, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftd3_64(long iopt, long n1, long n2, long n3, double scale, doublecomplex *x, long ldx1, long ldx2, double *y, long ldy1, long ldy2, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftd3 initializes the trigonometric weight and factor tables or computes the three-dimensional inverse Fast Fourier Transform of a three-dimensional double complex array. .Ve

```
                    N3-1  N2-1  N1-1
```

`Y(k1,k2,k3)` = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                    j3=0  j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**
  Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed.
- **LDX1 (input)**
  first dimension of X. LDX1 > = N1/2+1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. If X and Y are the same array, LDY1 = 2*LDX1 Else LDY1 > = 2*LDX1 and LDY1 is even Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least (MAX(N,2*N2,2*N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1/2+1)

  -6 = (LDX2 < N2)

  -7 = LDY1 not equal 2*LDX1 when X and Y are same array

  -8 = (LDY1 < 2*LDX1) or (LDY1 is odd) when X and Y are not same array

  -9 = (LDY2 < N2) or (LDY2 not equal LDX2) when X and Y are same array

  -10 = (LWORK not equal 0) and ((LWORK < `MAX(N,2*N2,2*N3)` + 16*N3)*NCPUS)

-11 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output subarray Y(1:LDY1, 1:N2, 1:N3) is overwritten.

NAME

zfftdm - initialize the trigonometric weight and factor tables or compute the one-dimensional inverse Fast Fourier Transform of a set of double complex data sequences stored in a two-dimensional array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTDM( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*)
 INTEGER IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY,*), TRIGS(*), WORK(*)

 SUBROUTINE ZFFTDM_64( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*)
 INTEGER*8 IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION Y(LDY,*), TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, M, [N], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X
 INTEGER :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: Y

 SUBROUTINE FFTM_64( IOPT, M, [N], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X
 INTEGER(8) :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
 REAL(8), DIMENSION(:,:) :: Y
```

C INTERFACE

#include <sunperf.h>

void zfftdm(int iopt, int m, int n, double scale, doublecomplex *x, int ldx, double *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftdm_64(long iopt, long m, long n, double scale, doublecomplex *x, long ldx, double *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftdm initializes the trigonometric weight and factor tables or computes the one-dimensional inverse Fast Fourier Transform of a set of double complex data sequences stored in a two-dimensional array: .Ve

```
                    N1-1
```

Y(k,l) = scale * SUM W*X(j,l)

```
                    j=0
.Ve
```

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform

W = `exp(isign*i*j*k*2*pi/N1)`

In complex-to-real transform of length N1, the (N1/2+1) complex input data points stored are the positive-frequency half of the spectrum of the Discrete Fourier Transform. The other half can be obtained through complex conjugation and therefore is not stored. Furthermore, due to symmetries the imaginary of the component of `X(0,0:N2-1)` and `X(N1/2,0:N2-1)` (if N1 is even in the latter) is assumed to be zero and is not referenced.

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = 1 computes inverse FFT
- **M (input)**
  Integer specifying length of the input sequences. M is most efficient when it is a product of small primes. M > = 0. Unchanged on exit.
- **N (input)**
  Integer specifying number of input sequences. N > = 0. Unchanged on exit.

- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX, N) that contains the sequences to be transformed stored in its columns in X(0:M/2, 0:N-1).
- **LDX (input)**
  Leading dimension of X. LDX > = (M/2+1) Unchanged on exit.
- **Y (output)**
  Y is a double precision array of dimensions (LDY, N) that contains the transform results of the input sequences in Y(0:M-1,0:N-1). X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = 2*LDX Else LDY > = M Unchanged on exit.
- **TRIGS (input/output)**
  double precision array of length 2*M that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of M. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1. Unchanged on exit.
- **WORK (output)**
  double precision array of dimension at least M. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0 or 1

  -2 = M < 0

  -3 = N < 0

  -4 = (LDX < M/2+1)

  -5 = (LDY < M) or (LDY not equal 2*LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < M)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

zfftf - compute the Fourier coefficients of a periodic sequence. The FFT operations are unnormalized, so a call of ZFFTF followed by a call of ZFFTB will multiply the input sequence by N.

SYNOPSIS

```
SUBROUTINE ZFFTF( N, X, WSAVE)
DOUBLE COMPLEX X(*)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE ZFFTF_64( N, X, WSAVE)
DOUBLE COMPLEX X(*)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE FFTF( [N], X, WSAVE)
COMPLEX(8), DIMENSION(:) :: X
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE FFTF_64( [N], X, WSAVE)
COMPLEX(8), DIMENSION(:) :: X
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void zfftf(int n, doublecomplex *x, double *wsave);

void zfftf_64(long n, doublecomplex *x, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. These subroutines are most efficient when N is a product of small primes. N > = 0.
- **X (input/output)**
  On entry, an array of length N containing the sequence to be transformed.
- **WSAVE (input)**
  On entry, WSAVE must be an array of dimension (4 * N + 15) or greater and must have been initialized by ZFFTI.

NAME

zffti - initialize the array WSAVE, which is used in both ZFFTF and ZFFTB.

SYNOPSIS

```
SUBROUTINE ZFFTI( N, WSAVE)
INTEGER N
DOUBLE PRECISION WSAVE(*)

SUBROUTINE ZFFTI_64( N, WSAVE)
INTEGER*8 N
DOUBLE PRECISION WSAVE(*)
```

F95 INTERFACE

```
SUBROUTINE ZFFTI( N, WSAVE)
INTEGER :: N
REAL(8), DIMENSION(:) :: WSAVE

SUBROUTINE ZFFTI_64( N, WSAVE)
INTEGER(8) :: N
REAL(8), DIMENSION(:) :: WSAVE
```

C INTERFACE

#include <sunperf.h>

void zffti(int n, double *wsave);

void zffti_64(long n, double *wsave);

ARGUMENTS

- **N (input)**
  Length of the sequence to be transformed. N > = 0.
- **WSAVE (input/output)**
  On entry, an array of dimension (4 * N + 15) or greater. ZFFTI needs to be called only once to initialize array WORK before calling ZFFTF and/or ZFFTB if N and WSAVE remain unchanged between these calls. Thus, subsequent transforms or inverse transforms of same size can be obtained faster than the first since they do not require initialization of the workspace.

NAME

zfftopt - compute the length of the closest fast FFT

SYNOPSIS

```
INTEGER FUNCTION ZFFTOPT( LEN)
INTEGER LEN

INTEGER*8 FUNCTION ZFFTOPT_64( LEN)
INTEGER*8 LEN
```

F95 INTERFACE

```
INTEGER FUNCTION ZFFTOPT( LEN)
INTEGER :: LEN

INTEGER(8) FUNCTION ZFFTOPT_64( LEN)
INTEGER(8) :: LEN
```

C INTERFACE

#include <sunperf.h>

int zfftopt(int len);

long zfftopt_64(long len);

PURPOSE

zfftopt computes the length of the closest fast FFT. Fast Fourier transform algorithms, including those used in Performance Library, work best with vector lengths that are products of small primes. For example, an FFT of length 32=2**5 will run faster than an FFT of prime length 31 because 32 is a product of small primes and 31 is not. If your application is such that you can taper or zero pad your vector to a larger length then this function may help you select a better length and run your FFT faster.

ZFFTOPT will return an integer no smaller than the input argument N that is the closest number that is the product of small primes. ZFFTOPT will return 16 for an input of N=16 and return 18=2*3*3 for an input of N=17.

Note that the length computed here is not guaranteed to be optimal, only to be a product of small primes. Also, the value returned may change as the underlying FFTs become capable of handling larger primes. For example, passing in N=51 to day will return 52=2*2*13 rather than 51=3*17 because the FFTs in Performance Li brary do not have fast radix 17 code. In the future, radix 17 code may be added and then N=51 will return 51.

ARGUMENTS

NAME

zfftz - initialize the trigonometric weight and factor tables or compute the Fast Fourier transform (forward or inverse) of a double complex sequence. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTZ( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX X(*), Y(*)
 INTEGER IOPT, N, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)

 SUBROUTINE ZFFTZ_64( IOPT, N, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK,
*      IERR)
 DOUBLE COMPLEX X(*), Y(*)
 INTEGER*8 IOPT, N, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK, [LWORK],
*       IERR)
 COMPLEX(8), DIMENSION(:) :: X, Y
 INTEGER :: IOPT, N, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT_64( IOPT, [N], [SCALE], X, Y, TRIGS, IFAC, WORK,
*       [LWORK], IERR)
 COMPLEX(8), DIMENSION(:) :: X, Y
 INTEGER(8) :: IOPT, N, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void zfftz(int iopt, int n, double scale, doublecomplex *x, doublecomplex *y, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftz_64(long iopt, long n, double scale, doublecomplex *x, doublecomplex *y, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftz initializes the trigonometric weight and factor tables or computes the Fast Fourier transform (forward or inverse) of a double complex sequence as follows: .Ve

```
                N-1
```

Y(k) = scale * SUM W*X(j)

```
                j=0
```

.Ve

where

k ranges from 0 to N-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N (input)**
  Integer specifying length of the input sequence X. N is most efficient when it is a product of small primes. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  On entry, X is a double complex array of dimension at least N that contains the sequence to be transformed.

- **Y (output)**
  Double complex array of dimension at least N that contains the transform results. X and Y may be the same array starting at the same memory location. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **TRIGS (input/output)**
  Double precision array of length 2*N that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of N. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls where IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least 2*N. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N < 0

  -3 = (LWORK is not 0) and (LWORK is less than 2*N)

  -4 = memory allocation for workspace failed

SEE ALSO

fft

NAME

zfftz2 - initialize the trigonometric weight and factor tables or compute the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTZ2( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)

 SUBROUTINE ZFFTZ2_64( IOPT, N1, N2, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER*8 IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT2( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X, Y
 INTEGER :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT2_64( IOPT, [N1], [N2], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X, Y
 INTEGER(8) :: IOPT, N1, N2, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void zfftz2(int iopt, int n1, int n2, double scale, doublecomplex *x, int ldx, doublecomplex *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftz2_64(long iopt, long n1, long n2, double scale, doublecomplex *x, long ldx, doublecomplex *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftz2 initializes the trigonometric weight and factor tables or computes the two-dimensional Fast Fourier Transform (forward or inverse) of a two-dimensional double complex array. In computing the two-dimensional FFT, one-dimensional FFTs are computed along the columns of the input array. One-dimensional FFTs are then computed along the rows of the intermediate results. .Ve

```
                    N2-1  N1-1
```

Y(k1,k2) = scale * SUM SUM W2*W1*X(j1,j2)

```
                    j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1 and k2 ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**

Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX, N2) that contains input data to be transformed.
- **LDX (input)**
  Leading dimension of X. LDX > = N1 Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY, N2) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = LDX Else LDY > = N1 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 2*128 that contains the factors of N1 and N2. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least 2*MAX(N1,N2)*NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = (LDX < N1)

  -5 = (LDY < N1) or (LDY not equal LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < 2*MAX(N1,N2)*NCPUS)

  -7 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, entire output array Y(1:LDY, 1:N2) is overwritten.

NAME

zfftz3 - initialize the trigonometric weight and factor tables or compute the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTZ3( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1,
*      LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX1,LDX2,*), Y(LDY1,LDY2,*)
 INTEGER IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)

 SUBROUTINE ZFFTZ3_64( IOPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y,
*      LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX1,LDX2,*), Y(LDY1,LDY2,*)
 INTEGER*8 IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFT3( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1], LDX2,
*       Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: X, Y
 INTEGER :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFT3_64( IOPT, [N1], [N2], [N3], [SCALE], X, [LDX1],
*       LDX2, Y, [LDY1], LDY2, TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:,:) :: X, Y
 INTEGER(8) :: IOPT, N1, N2, N3, LDX1, LDX2, LDY1, LDY2, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void zfftz3(int iopt, int n1, int n2, int n3, double scale, doublecomplex *x, int ldx1, int ldx2, doublecomplex *y, int ldy1, int ldy2, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftz3_64(long iopt, long n1, long n2, long n3, double scale, doublecomplex *x, long ldx1, long ldx2, doublecomplex *y, long ldy1, long ldy2, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftz3 initializes the trigonometric weight and factor tables or computes the three-dimensional Fast Fourier Transform (forward or inverse) of a three-dimensional double complex array. .Ve

```
                        N3-1  N2-1  N1-1
```

Y(k1,k2,k3) = scale * SUM SUM SUM W3*W2*W1*X(j1,j2,j3)

```
                        j3=0  j2=0  j1=0
```

.Ve

where

k1 ranges from 0 to N1-1; k2 ranges from 0 to N2-1 and k3 ranges from 0 to N3-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W1 = `exp(isign*i*j1*k1*2*pi/N1)`

W2 = `exp(isign*i*j2*k2*2*pi/N2)`

W3 = `exp(isign*i*j3*k3*2*pi/N3)`

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **N1 (input)**
  Integer specifying length of the transform in the first dimension. N1 is most efficient when it is a product of small primes. N1 > = 0. Unchanged on exit.
- **N2 (input)**

Integer specifying length of the transform in the second dimension. N2 is most efficient when it is a product of small primes. N2 > = 0. Unchanged on exit.

- **N3 (input)**
  Integer specifying length of the transform in the third dimension. N3 is most efficient when it is a product of small primes. N3 > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.
- **X (input)**
  X is a double complex array of dimensions (LDX1, LDX2, N3) that contains input data to be transformed.
- **LDX1 (input)**
  first dimension of X. LDX1 > = N1 Unchanged on exit.
- **LDX2 (input)**
  second dimension of X. LDX2 > = N2 Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY1, LDY2, N3) that contains the transform results. X and Y can be the same array starting at the same memory location, in which case the input data are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY1 (input)**
  first dimension of Y. If X and Y are the same array, LDY1 = LDX1 Else LDY1 > = N1 Unchanged on exit.
- **LDY2 (input)**
  second dimension of Y. If X and Y are the same array, LDY2 = LDX2 Else LDY2 > = N2 Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*(N1+N2+N3) that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 3*128 that contains the factors of N1, N2 and N3. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least (2*MAX(N,N2,N3) + 16*N3) * NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = N1 < 0

  -3 = N2 < 0

  -4 = N3 < 0

  -5 = (LDX1 < N1)

  -6 = (LDX2 < N2)

  -7 = (LDY1 < N1) or (LDY1 not equal LDX1 when X and Y are same array)

  -8 = (LDY2 < N2) or (LDY2 not equal LDX2 when X and Y are same array)

-9 = (LWORK not equal 0) and (LWORK < (2*MAX(N,N2,N3) + 16*N3) * NCPUS)

-10 = memory allocation failed

SEE ALSO

fft

CAUTIONS

On exit, output subarray Y(1:LDY1, 1:N2, 1:N3) is overwritten.

NAME

zfftzm - initialize the trigonometric weight and factor tables or compute the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional double complex array. =head1 SYNOPSIS

```
 SUBROUTINE ZFFTZM( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS, IFAC,
*      WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)

 SUBROUTINE ZFFTZM_64( IOPT, M, N, SCALE, X, LDX, Y, LDY, TRIGS,
*      IFAC, WORK, LWORK, IERR)
 DOUBLE COMPLEX X(LDX,*), Y(LDY,*)
 INTEGER*8 IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER*8 IFAC(*)
 DOUBLE PRECISION SCALE
 DOUBLE PRECISION TRIGS(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE FFTM( IOPT, [M], [N], [SCALE], X, [LDX], Y, [LDY], TRIGS,
*       IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X, Y
 INTEGER :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER, DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK

 SUBROUTINE FFTM_64( IOPT, [M], [N], [SCALE], X, [LDX], Y, [LDY],
*       TRIGS, IFAC, WORK, [LWORK], IERR)
 COMPLEX(8), DIMENSION(:,:) :: X, Y
 INTEGER(8) :: IOPT, M, N, LDX, LDY, LWORK, IERR
 INTEGER(8), DIMENSION(:) :: IFAC
 REAL(8) :: SCALE
 REAL(8), DIMENSION(:) :: TRIGS, WORK
```

C INTERFACE

#include <sunperf.h>

void zfftzm(int iopt, int m, int n, double scale, doublecomplex *x, int ldx, doublecomplex *y, int ldy, double *trigs, int *ifac, double *work, int lwork, int *ierr);

void zfftzm_64(long iopt, long m, long n, double scale, doublecomplex *x, long ldx, doublecomplex *y, long ldy, double *trigs, long *ifac, double *work, long lwork, long *ierr);

PURPOSE

zfftzm initializes the trigonometric weight and factor tables or computes the one-dimensional Fast Fourier Transform (forward or inverse) of a set of data sequences stored in a two-dimensional double complex array: .Ve

```
        N1-1
```

Y(k,l) = SUM W*X(j,l)

```
        j=0
```

.Ve

where

k ranges from 0 to N1-1 and l ranges from 0 to N2-1

i = `sqrt(-1)`

isign = 1 for inverse transform or -1 for forward transform

W = `exp(isign*i*j*k*2*pi/N1)` .Ve

ARGUMENTS

- **IOPT (input)**
  Integer specifying the operation to be performed:

  IOPT = 0 computes the trigonometric weight table and factor table

  IOPT = -1 computes forward FFT

  IOPT = +1 computes inverse FFT
- **M (input)**
  Integer specifying length of the input sequences. M is most efficient when it is a product of small primes. M > = 0. Unchanged on exit.
- **N (input)**
  Integer specifying number of input sequences. N > = 0. Unchanged on exit.
- **SCALE (input)**
  Double precision scalar by which transform results are scaled. Unchanged on exit.

- **X (input)**
  X is a double complex array of dimensions (LDX, N) that contains the sequences to be transformed stored in its columns.
- **LDX (input)**
  Leading dimension of X. LDX > = M Unchanged on exit.
- **Y (output)**
  Y is a double complex array of dimensions (LDY, N) that contains the transform results of the input sequences. X and Y can be the same array starting at the same memory location, in which case the input sequences are overwritten by their transform results. Otherwise, it is assumed that there is no overlap between X and Y in memory.
- **LDY (input)**
  Leading dimension of Y. If X and Y are the same array, LDY = LDX Else LDY > = M Unchanged on exit.
- **TRIGS (input/output)**
  Double precision array of length 2*M that contains the trigonometric weights. The weights are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **IFAC (input/output)**
  Integer array of dimension at least 128 that contains the factors of M. The factors are computed when the routine is called with IOPT = 0 and they are used in subsequent calls when IOPT = 1 or IOPT = -1. Unchanged on exit.
- **WORK (output)**
  Double precision array of dimension at least 2*M*NCPUS where NCPUS is the number of threads used to execute the routine. The user can also choose to have the routine allocate its own workspace (see LWORK).
- **LWORK (input)**
  Integer specifying workspace size. If LWORK = 0, the routine will allocate its own workspace.
- **IERR (output)**
  On exit, integer IERR has one of the following values:

  0 = normal return

  -1 = IOPT is not 0, 1 or -1

  -2 = M < 0

  -3 = N < 0

  -4 = (LDX < M)

  -5 = (LDY < M) or (LDY not equal LDX when X and Y are same array)

  -6 = (LWORK not equal 0) and (LWORK < 2*M*NCPUS)

  -7 = memory allocation failed

SEE ALSO

fft

NAME

zgbbrd - reduce a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation

---

SYNOPSIS

```
 SUBROUTINE ZGBBRD( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q, LDQ,
*      PT, LDPT, C, LDC, WORK, RWORK, INFO)
 CHARACTER * 1 VECT
 DOUBLE COMPLEX AB(LDAB,*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
 INTEGER M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 DOUBLE PRECISION D(*), E(*), RWORK(*)

 SUBROUTINE ZGBBRD_64( VECT, M, N, NCC, KL, KU, AB, LDAB, D, E, Q,
*      LDQ, PT, LDPT, C, LDC, WORK, RWORK, INFO)
 CHARACTER * 1 VECT
 DOUBLE COMPLEX AB(LDAB,*), Q(LDQ,*), PT(LDPT,*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 DOUBLE PRECISION D(*), E(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GBBRD( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D, E,
*       Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Q, PT, C
 INTEGER :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, RWORK

 SUBROUTINE GBBRD_64( VECT, [M], [N], [NCC], KL, KU, AB, [LDAB], D,
*       E, Q, [LDQ], PT, [LDPT], C, [LDC], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Q, PT, C
 INTEGER(8) :: M, N, NCC, KL, KU, LDAB, LDQ, LDPT, LDC, INFO
 REAL(8), DIMENSION(:) :: D, E, RWORK
```

C INTERFACE

#include <sunperf.h>

void zgbbrd(char *vect, int m, int n, int ncc, int kl, int ku, doublecomplex *ab, int ldab, double *d, double *e, doublecomplex *q, int ldq, doublecomplex *pt, int ldpt, doublecomplex *c, int ldc, int *info);

void zgbbrd_64(char *vect, long m, long n, long ncc, long kl, long ku, doublecomplex *ab, long ldab, double *d, double *e, doublecomplex *q, long ldq, doublecomplex *pt, long ldpt, doublecomplex *c, long ldc, long *info);

PURPOSE

zgbbrd reduces a complex general m-by-n band matrix A to real upper bidiagonal form B by a unitary transformation: Q' * A * P = B.

The routine computes B, and optionally forms Q or P', or computes Q'*C for a given matrix C.

ARGUMENTS

- **VECT (input/output)**
  Specifies whether or not the matrices Q and P' are to be formed. = 'N': do not form Q or P';

  = 'Q': form Q only;

  = 'P': form P' only;

  = 'B': form both.
- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NCC (input)**
  The number of columns of the matrix C. NCC >= 0.
- **KL (input)**
  The number of subdiagonals of the matrix A. KL >= 0.
- **KU (input)**
  The number of superdiagonals of the matrix A. KU >= 0.
- **AB (input/output)**
  On entry, the m-by-n band matrix A, stored in rows 1 to KL+KU+1. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl). On exit, A is overwritten by values generated during the reduction.
- **LDAB (input)**
  The leading dimension of the array A. LDAB >= KL+KU+1.
- **D (output)**
  The diagonal elements of the bidiagonal matrix B.
- **E (output)**

The superdiagonal elements of the bidiagonal matrix B.

- **Q (output)**
  If VECT = 'Q' or 'B', the m-by-m unitary matrix Q. If VECT = 'N' or 'P', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,M)` if VECT = 'Q' or 'B'; LDQ >= 1 otherwise.
- **PT (output)**
  If VECT = 'P' or 'B', the n-by-n unitary matrix P'. If VECT = 'N' or 'Q', the array PT is not referenced.
- **LDPT (input)**
  The leading dimension of the array PT. LDPT >= `max(1,N)` if VECT = 'P' or 'B'; LDPT >= 1 otherwise.
- **C (input/output)**
  On entry, an m-by-ncc matrix C. On exit, C is overwritten by Q'*C. C is not referenced if NCC = 0.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)` if NCC > 0; LDC >= 1 if NCC = 0.
- **WORK (workspace)**
  `dimension(MAX(M,N))`
- **RWORK (workspace)**
  `dimension(MAX(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

zgbcon - estimate the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm,

---

SYNOPSIS

```
 SUBROUTINE ZGBCON( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NSUB, NSUPER, LDA, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGBCON_64( NORM, N, NSUB, NSUPER, A, LDA, IPIVOT, ANORM,
*       RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, LDA, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBCON( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT, ANORM,
*       RCOND, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, NSUB, NSUPER, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GBCON_64( NORM, [N], NSUB, NSUPER, A, [LDA], IPIVOT,
*       ANORM, RCOND, [WORK], [WORK2], [INFO])
```

```
CHARACTER(LEN=1) :: NORM
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgbcon(char norm, int n, int nsub, int nsuper, doublecomplex *a, int lda, int *ipivot, double anorm, double *rcond, int *info);

void zgbcon_64(char norm, long n, long nsub, long nsuper, doublecomplex *a, long lda, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zgbcon estimates the reciprocal of the condition number of a complex general band matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.

- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  dimension (N)
- **INFO (output)**

```
= 0:  successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgbequ - compute row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE ZGBEQU( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC, ROWCN,
*      COLCN, AMAX, INFO)
 DOUBLE COMPLEX A(LDA,*)
 INTEGER M, N, NSUB, NSUPER, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION ROWSC(*), COLSC(*)

 SUBROUTINE ZGBEQU_64( M, N, NSUB, NSUPER, A, LDA, ROWSC, COLSC,
*      ROWCN, COLCN, AMAX, INFO)
 DOUBLE COMPLEX A(LDA,*)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GBEQU( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*       ROWCN, COLCN, AMAX, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC

 SUBROUTINE GBEQU_64( [M], [N], NSUB, NSUPER, A, [LDA], ROWSC, COLSC,
*       ROWCN, COLCN, AMAX, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
```

C INTERFACE

#include <sunperf.h>

void zgbequ(int m, int n, int nsub, int nsuper, doublecomplex *a, int lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, int *info);

void zgbequ_64(long m, long n, long nsub, long nsuper, doublecomplex *a, long lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, long *info);

PURPOSE

zgbequ computes row and column scalings intended to equilibrate an M-by-N band matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input)**
  The band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **ROWSC (output)**
  If INFO = 0, or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, and i is

  < = M: the i-th row of A is exactly zero

  > M: the (i-M)-th column of A is exactly zero

NAME

zgbmv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE ZGBMV( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER M, N, NSUB, NSUPER, LDA, INCX, INCY

 SUBROUTINE ZGBMV_64( TRANSA, M, N, NSUB, NSUPER, ALPHA, A, LDA, X,
*      INCX, BETA, Y, INCY)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 M, N, NSUB, NSUPER, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE GBMV( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, NSUB, NSUPER, LDA, INCX, INCY

 SUBROUTINE GBMV_64( [TRANSA], [M], [N], NSUB, NSUPER, ALPHA, A, [LDA],
*       X, [INCX], BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zgbmv(char transa, int m, int n, int nsub, int nsuper, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *x, int incx, doublecomplex beta, doublecomplex *y, int incy);

void zgbmv_64(char transa, long m, long n, long nsub, long nsuper, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *x, long incx, doublecomplex beta, doublecomplex *y, long incy);

PURPOSE

zgbmv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with nsub sub-diagonals and nsuper super-diagonals.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*conjg(A')*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M must be at least zero. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N must be at least zero. Unchanged on exit.
- **NSUB (input)**
  On entry, NSUB specifies the number of sub-diagonals of the matrix A. NSUB must satisfy 0 .le. NSUB. Unchanged on exit.
- **NSUPER (input)**
  On entry, NSUPER specifies the number of super-diagonals of the matrix A. NSUPER must satisfy 0 .le. NSUPER. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading (nsub + nsuper + 1) by n part of the array A must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (nsuper + 1) of the array, the first super-diagonal starting at position 2 in row nsuper, the first sub-diagonal starting at position 1 in row (nsuper + 2), and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left nsuper by nsuper triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
```

```
      K = NSUPER + 1 - J
      DO 10, I = MAX( 1, J - NSUPER ), MIN( M, J + NSUB )
        A( K + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA must be at least (nsub + nsuper + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zgbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZGBRFS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF, LDAF,
*      IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGBRFS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*      LDAF, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBRFS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GBRFS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
```

```
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgbrfs(char transa, int n, int nsub, int nsuper, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zgbrfs_64(char transa, long n, long nsub, long nsuper, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zgbrfs improves the computed solution to a system of linear equations when the coefficient matrix is banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The original band matrix A, stored in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(n,j+kl).
- **LDA (input)**
  The leading dimension of the array A. LDA >= NSUB+NSUPER+1.
- **AF (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during

the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB*NSUPER+1.
- **IPIVOT (input)**
  The pivot indices from CGBTRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CGBTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgbsv - compute the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE ZGBSV( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE ZGBSV_64( N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B, LDB,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSV( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GBSV_64( [N], NSUB, NSUPER, [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgbsv(int n, int nsub, int nsuper, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zgbsv_64(long n, long nsub, long nsuper, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zgbsv computes the solution to a complex system of linear equations A * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = L * U, where L is a product of permutation and unit lower triangular matrices with KL subdiagonals, and U is upper triangular with KL+KU superdiagonals. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUB+NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+NSUB) On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +       *    *    *   u14  u25  u36
 *    *    +    +    +    +       *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *      m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *      m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

zgbsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZGBSVX( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, AF,
*      LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGBSVX_64( FACT, TRANSA, N, NSUB, NSUPER, NRHS, A, LDA,
*      AF, LDAF, IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND,
*      FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GBSVX( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

```
 SUBROUTINE GBSVX_64( FACT, [TRANSA], [N], NSUB, NSUPER, [NRHS], A,
*       [LDA], AF, [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX],
*       RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgbsvx(char fact, char transa, int n, int nsub, int nsuper, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, char equed, double *rowsc, double *colsc, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zgbsvx_64(char fact, char transa, long n, long nsub, long nsuper, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, char equed, double *rowsc, double *colsc, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zgbsvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed by this subroutine:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
 Whether or not the system will be equilibrated depends on the
 scaling of the matrix A, but if equilibration is used, A is
 overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
 or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = L * U,

 where L is a product of permutation and unit lower triangular
 matrices with KL subdiagonals, and U is upper triangular with
 KL+KU superdiagonals.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine

precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```
- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows 1 to NSUB+NSUPER+1. The j-th column of A is stored in the j-th column of the array A as follows: `A(NSUPER+1+i-j,j)` = `A(i,j)` for `max(1,j-NSUPER)` < =i < =min(N,j+kl)

  If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NSUB+NSUPER+1.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains details of the LU factorization of the band matrix

A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

If FACT = 'N', then AF is an output argument and on exit returns details of the LU factorization of A.

If FACT = 'E', then AF is an output argument and on exit returns details of the LU factorization of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = 2*NSUB+NSUPER+1.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = L*U as computed by CGBTRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = L*U of the equilibrated matrix A.
- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.
- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(2*N)
- **WORK2 (workspace)**
  dimension(N) On exit, WORK2(1) contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If WORK2(1) is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then WORK2(1) contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zgbtf2 - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE ZGBTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO)
DOUBLE COMPLEX AB(LDAB,*)
INTEGER M, N, KL, KU, LDAB, INFO
INTEGER IPIV(*)

SUBROUTINE ZGBTF2_64( M, N, KL, KU, AB, LDAB, IPIV, INFO)
DOUBLE COMPLEX AB(LDAB,*)
INTEGER*8 M, N, KL, KU, LDAB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE GBTF2( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER :: M, N, KL, KU, LDAB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE GBTF2_64( [M], [N], KL, KU, AB, [LDAB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER(8) :: M, N, KL, KU, LDAB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zgbtf2(int m, int n, int kl, int ku, doublecomplex *ab, int ldab, int *ipiv, int *info);

void zgbtf2_64(long m, long n, long kl, long ku, doublecomplex *ab, long ldab, long *ipiv, long *info);

PURPOSE

zgbtf2 computes an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **KL (input)**
  The number of subdiagonals within the band of A. KL >= 0.
- **KU (input)**
  The number of superdiagonals within the band of A. KU >= 0.
- **AB (input/output)**
  On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows: `AB(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1. See below for further details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= 2*KL+KU+1.
- **IPIV (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, KL = 2, KU = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U, because of fill-in resulting from the row

interchanges.

NAME

zgbtrf - compute an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE ZGBTRF( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER M, N, NSUB, NSUPER, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGBTRF_64( M, N, NSUB, NSUPER, A, LDA, IPIVOT, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 M, N, NSUB, NSUPER, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GBTRF( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GBTRF_64( [M], [N], NSUB, NSUPER, A, [LDA], IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, NSUB, NSUPER, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgbtrf(int m, int n, int nsub, int nsuper, doublecomplex *a, int lda, int *ipivot, int *info);

void zgbtrf_64(long m, long n, long nsub, long nsuper, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zgbtrf computes an LU factorization of a complex m-by-n band matrix A using partial pivoting with row interchanges.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB >= 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER >= 0.
- **A (input/output)**
  On entry, the matrix A in band storage, in rows NSUB+1 to 2*NSUB+NSUPER+1; rows 1 to NSUB of the array need not be set. The j-th column of A is stored in the j-th column of the array A as follows: `A(kl+ku+1+i-j,j)` = `A(i,j)` for `max(1,j-ku)` <=i <=min(m,j+kl)

  On exit, details of the factorization: U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1. See below for further details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= 2*NSUB+NSUPER+1.
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = +i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, NSUB = 2, NSUPER = 1:

On entry: On exit:

```
 *    *    *    +    +    +        *    *    *   u14  u25  u36
 *    *    +    +    +    +        *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56       *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66      u11  u22  u33  u44  u55  u66
a21  a32  a43  a54  a65   *       m21  m32  m43  m54  m65   *
a31  a42  a53  a64   *    *       m31  m42  m53  m64   *    *
```

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

zgbtrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF

SYNOPSIS

```
 SUBROUTINE ZGBTRS( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE ZGBTRS_64( TRANSA, N, NSUB, NSUPER, NRHS, A, LDA, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GBTRS( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GBTRS_64( [TRANSA], [N], NSUB, NSUPER, [NRHS], A, [LDA],
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NSUB, NSUPER, NRHS, LDA, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgbtrs(char transa, int n, int nsub, int nsuper, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zgbtrs_64(char transa, long n, long nsub, long nsuper, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zgbtrs solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general band matrix A using the LU factorization computed by CGBTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NSUB (input)**
  The number of subdiagonals within the band of A. NSUB > = 0.
- **NSUPER (input)**
  The number of superdiagonals within the band of A. NSUPER > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  Details of the LU factorization of the band matrix A, as computed by CGBTRF. U is stored as an upper triangular band matrix with NSUB+NSUPER superdiagonals in rows 1 to NSUB+NSUPER+1, and the multipliers used during the factorization are stored in rows NSUB+NSUPER+2 to 2*NSUB+NSUPER+1.
- **LDA (input)**
  The leading dimension of the array A. LDA > = 2*NSUB+NSUPER+1.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgebak - form the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL

SYNOPSIS

```
 SUBROUTINE ZGEBAK( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 DOUBLE COMPLEX V(LDV,*)
 INTEGER N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION SCALE(*)

 SUBROUTINE ZGEBAK_64( JOB, SIDE, N, ILO, IHI, SCALE, M, V, LDV,
*      INFO)
 CHARACTER * 1 JOB, SIDE
 DOUBLE COMPLEX V(LDV,*)
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBAK( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX(8), DIMENSION(:,:) :: V
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: SCALE

 SUBROUTINE GEBAK_64( JOB, SIDE, [N], ILO, IHI, SCALE, [M], V, [LDV],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX(8), DIMENSION(:,:) :: V
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void zgebak(char job, char side, int n, int ilo, int ihi, double *scale, int m, doublecomplex *v, int ldv, int *info);

void zgebak_64(char job, char side, long n, long ilo, long ihi, double *scale, long m, doublecomplex *v, long ldv, long *info);

PURPOSE

zgebak forms the right or left eigenvectors of a complex general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by CGEBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required: = 'N', do nothing, return immediately; = 'P', do backward transformation for permutation only; = 'S', do backward transformation for scaling only; = 'B', do backward transformations for both permutation and scaling. JOB must be the same as the argument JOB supplied to CGEBAL.
- **SIDE (input)**

  ```
  = 'R':  V contains right eigenvectors;

  = 'L':  V contains left eigenvectors.
  ```
- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integer ILO determined by CGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  The integer IHI determined by CGEBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **SCALE (input)**
  Details of the permutation and scaling factors, as returned by CGEBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.
- **V (input/output)**
  On entry, the matrix of right or left eigenvectors to be transformed, as returned by CHSEIN or CTREVC. On exit, V is overwritten by the transformed eigenvectors.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

zgebal - balance a general complex matrix A

SYNOPSIS

```
SUBROUTINE ZGEBAL( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, ILO, IHI, INFO
DOUBLE PRECISION SCALE(*)

SUBROUTINE ZGEBAL_64( JOB, N, A, LDA, ILO, IHI, SCALE, INFO)
CHARACTER * 1 JOB
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, ILO, IHI, INFO
DOUBLE PRECISION SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE GEBAL( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, ILO, IHI, INFO
REAL(8), DIMENSION(:) :: SCALE

SUBROUTINE GEBAL_64( JOB, [N], A, [LDA], ILO, IHI, SCALE, [INFO])
CHARACTER(LEN=1) :: JOB
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, ILO, IHI, INFO
REAL(8), DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void zgebal(char job, int n, doublecomplex *a, int lda, int *ilo, int *ihi, double *scale, int *info);

void zgebal_64(char job, long n, doublecomplex *a, long lda, long *ilo, long *ihi, double *scale, long *info);

PURPOSE

zgebal balances a general complex matrix A. This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to ILO-1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, SCALE(I)  = 1.0
for i  = 1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  ILO and IHI are set to integers such that on exit `A(i,j)` = 0 if i > j and j = 1,...,ILO-1 or I = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **SCALE (output)**
  Details of the permutations and scaling factors applied to A. If `P(j)` is the index of the row and column interchanged with row and column j and `D(j)` is the scaling factor applied to row and column j, then `SCALE(j)` = `P(j)` for j = 1,...,ILO-1 = `D(j)` for j = ILO,...,IHI = `P(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

FURTHER DETAILS

The permutations consist of row and column interchanges which put the matrix in the form

```
          ( T1   X   Y  )
P A P  = (  0   B   Z  )
          (  0   0   T2 )
```

where T1 and T2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ILO and IHI mark the starting and ending columns of the submatrix B. Balancing consists of applying a diagonal similarity transformation `inv(D)` * B * D to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

```
( T1     X*D          Y    )
(  0  inv(D)*B*D  inv(D)*Z ).
(  0      0           T2   )
```

Information about the permutations P and the diagonal matrix D is returned in the vector SCALE.

This subroutine is based on the EISPACK routine CBAL.

Modified by Tzu-Yi Chen, Computer Science Division, University of California at Berkeley, USA

NAME

zgebrd - reduce a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation

---

SYNOPSIS

```
 SUBROUTINE ZGEBRD( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAUQ(*), TAUP(*), WORK(*)
 INTEGER M, N, LDA, LWORK, INFO
 DOUBLE PRECISION D(*), E(*)

 SUBROUTINE ZGEBRD_64( M, N, A, LDA, D, E, TAUQ, TAUP, WORK, LWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAUQ(*), TAUP(*), WORK(*)
 INTEGER*8 M, N, LDA, LWORK, INFO
 DOUBLE PRECISION D(*), E(*)
```

F95 INTERFACE

```
 SUBROUTINE GEBRD( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUQ, TAUP, WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, LWORK, INFO
 REAL(8), DIMENSION(:) :: D, E

 SUBROUTINE GEBRD_64( [M], [N], A, [LDA], D, E, TAUQ, TAUP, [WORK],
*       [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUQ, TAUP, WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 REAL(8), DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void zgebrd(int m, int n, doublecomplex *a, int lda, double *d, double *e, doublecomplex *tauq, doublecomplex *taup, int *info);

void zgebrd_64(long m, long n, doublecomplex *a, long lda, double *d, double *e, doublecomplex *tauq, doublecomplex *taup, long *info);

PURPOSE

zgebrd reduces a general complex M-by-N matrix A to upper or lower bidiagonal form B by a unitary transformation: Q**H * A * P = B.

If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.

ARGUMENTS

- **M (input)**
  The number of rows in the matrix A. M > = 0.
- **N (input)**
  The number of columns in the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N general matrix to be reduced. On exit, if m > = n, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array TAUQ, represent the unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array TAUP, represent the unitary matrix P as a product of elementary reflectors; if m < n, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array TAUQ, represent the unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the unitary matrix P as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **D (output)**
  The diagonal elements of the bidiagonal matrix B: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the bidiagonal matrix B: if m > = n, `E(i)` = `A(i,i+1)` for i = 1,2,...,n-1; if m < n, `E(i)` = `A(i+1,i)` for i = 1,2,...,m-1.
- **TAUQ (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q. See Further Details.
- **TAUP (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix P. See Further Details.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,M,N). For optimum performance LWORK > = (M+N)*NB,

where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrices Q and P are represented as products of elementary reflectors:

If m >= n,

```
Q  = H(1) H(2) . . . H(n)  and  P  = G(1) G(2) . . . G(n-1)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are complex scalars, and v and u are complex vectors; `v(1:i-1)` = 0, `v(i)` = 1, and `v(i+1:m)` is stored on exit in A(i+1:m,i); `u(1:i)` = 0, `u(i+1)` = 1, and `u(i+2:n)` is stored on exit in A(i,i+2:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

If m < n,

```
Q  = H(1) H(2) . . . H(m-1)  and  P  = G(1) G(2) . . . G(m)
```

Each `H(i)` and `G(i)` has the form:

```
H(i)  = I - tauq * v * v'  and G(i)  = I - taup * u * u'
```

where tauq and taup are complex scalars, and v and u are complex vectors; `v(1:i)` = 0, `v(i+1)` = 1, and `v(i+2:m)` is stored on exit in A(i+2:m,i); `u(1:i-1)` = 0, `u(i)` = 1, and `u(i+1:n)` is stored on exit in A(i,i+1:n); tauq is stored in `TAUQ(i)` and taup in TAUP(i).

The contents of A on exit are illustrated by the following examples:

m = 6 and n = 5 (m > n): m = 5 and n = 6 (m < n):

```
(  d   e   u1  u1  u1 )          (  d   u1  u1  u1  u1  u1 )
(  v1  d   e   u2  u2 )          (  e   d   u2  u2  u2  u2 )
(  v1  v2  d   e   u3 )          (  v1  e   d   u3  u3  u3 )
(  v1  v2  v3  d   e  )          (  v1  v2  e   d   u4  u4 )
(  v1  v2  v3  v4  d  )          (  v1  v2  v3  e   d   u5 )
(  v1  v2  v3  v4  v5 )
```

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining H(i), and ui an element of the vector defining G(i).

NAME

zgecon - estimate the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF

SYNOPSIS

```
 SUBROUTINE ZGECON( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGECON_64( NORM, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GECON( NORM, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GECON_64( NORM, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgecon(char norm, int n, doublecomplex *a, int lda, double anorm, double *rcond, int *info);

void zgecon_64(char norm, long n, doublecomplex *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

zgecon estimates the reciprocal of the condition number of a general complex matrix A, in either the 1-norm or the infinity-norm, using the LU factorization computed by CGETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgeequ - compute row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number

---

SYNOPSIS

```
 SUBROUTINE ZGEEQU( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN, AMAX,
*      INFO)
 DOUBLE COMPLEX A(LDA,*)
 INTEGER M, N, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION ROWSC(*), COLSC(*)

 SUBROUTINE ZGEEQU_64( M, N, A, LDA, ROWSC, COLSC, ROWCN, COLCN,
*      AMAX, INFO)
 DOUBLE COMPLEX A(LDA,*)
 INTEGER*8 M, N, LDA, INFO
 DOUBLE PRECISION ROWCN, COLCN, AMAX
 DOUBLE PRECISION ROWSC(*), COLSC(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEQU( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*       AMAX, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC

 SUBROUTINE GEEQU_64( [M], [N], A, [LDA], ROWSC, COLSC, ROWCN, COLCN,
*       AMAX, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, INFO
 REAL(8) :: ROWCN, COLCN, AMAX
 REAL(8), DIMENSION(:) :: ROWSC, COLSC
```

C INTERFACE

#include <sunperf.h>

void zgeequ(int m, int n, doublecomplex *a, int lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, int *info);

void zgeequ_64(long m, long n, doublecomplex *a, long lda, double *rowsc, double *colsc, double *rowcn, double *colcn, double *amax, long *info);

PURPOSE

zgeequ computes row and column scalings intended to equilibrate an M-by-N matrix A and reduce its condition number. R returns the row scale factors and C the column scale factors, chosen to try to make the largest element in each row and column of the matrix B with elements `B(i,j)=R(i)*A(i,j)*C(j)` have absolute value 1.

`R(i)` and `C(j)` are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input)**
  The M-by-N matrix whose equilibration factors are to be computed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **ROWSC (output)**
  If INFO = 0 or INFO > M, ROWSC contains the row scale factors for A.
- **COLSC (output)**
  If INFO = 0, COLSC contains the column scale factors for A.
- **ROWCN (output)**
  If INFO = 0 or INFO > M, ROWCN contains the ratio of the smallest `ROWSC(i)` to the largest ROWSC(i). If ROWCN >= 0.1 and AMAX is neither too large nor too small, it is not worth scaling by ROWSC.
- **COLCN (output)**
  If INFO = 0, COLCN contains the ratio of the smallest `COLSC(i)` to the largest COLSC(i). If COLCN >= 0.1, it is not worth scaling by COLSC.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: if INFO = i, and i is

< = M: the i-th row of A is exactly zero

> M: the (i-M)-th column of A is exactly zero

NAME

zgees - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE ZGEES( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, W, Z, LDZ,
*      WORK, LDWORK, WORK2, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 DOUBLE COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL WORK3(*)
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGEES_64( JOBZ, SORTEV, SELECT, N, A, LDA, NOUT, W, Z,
*      LDZ, WORK, LDWORK, WORK2, WORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV
 DOUBLE COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 WORK3(*)
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEES( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, W, Z,
*       [LDZ], [WORK], [LDWORK], [WORK2], [WORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: WORK3
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GEES_64( JOBZ, SORTEV, SELECT, [N], A, [LDA], NOUT, W, Z,
*       [LDZ], [WORK], [LDWORK], [WORK2], [WORK3], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, SORTEV
COMPLEX(8), DIMENSION(:) :: W, WORK
COMPLEX(8), DIMENSION(:,:) :: A, Z
INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
LOGICAL(8) :: SELECT
LOGICAL(8), DIMENSION(:) :: WORK3
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgees(char jobz, char sortev, logical(*select)(COMPLEX*16), int n, doublecomplex *a, int lda, int *nout, doublecomplex *w, doublecomplex *z, int ldz, int *info);

void zgees_64(char jobz, char sortev, logical(*select)(COMPLEX*16), long n, doublecomplex *a, long lda, long *nout, doublecomplex *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zgees computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**H).

Optionally, it also orders the eigenvalues on the diagonal of the Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A complex matrix is in Schur form if it is upper triangular.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.
  ```

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered:

  ```
  = 'S': Eigenvalues are ordered (see SELECT).
  ```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to order to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. The eigenvalue `W(j)` is selected if `SELECT(W(j))` is true.

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**

On entry, the N-by-N matrix A. On exit, A has been overwritten by its Schur form T.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues for which SELECT is true.
- **W (output)**
  W contains the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T.
- **Z (output)**
  If JOBZ = 'V', Z contains the unitary matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1; if JOBZ = 'V', LDZ >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **WORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.

> 0: if INFO  = i, and i is

< = N:  the QR algorithm failed to compute all the
```

eigenvalues; elements 1:ILO-1 and i+1:N of W contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the matrix which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT = .TRUE.. This could also be caused by underflow due to scaling.

NAME

zgeesx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z

SYNOPSIS

```
 SUBROUTINE ZGEESX( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT, W,
*      Z, LDZ, RCONE, RCONV, WORK, LDWORK, WORK2, BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 DOUBLE COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL SELECT
 LOGICAL BWORK3(*)
 DOUBLE PRECISION RCONE, RCONV
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGEESX_64( JOBZ, SORTEV, SELECT, SENSE, N, A, LDA, NOUT,
*      W, Z, LDZ, RCONE, RCONV, WORK, LDWORK, WORK2, BWORK3, INFO)
 CHARACTER * 1 JOBZ, SORTEV, SENSE
 DOUBLE COMPLEX A(LDA,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL*8 SELECT
 LOGICAL*8 BWORK3(*)
 DOUBLE PRECISION RCONE, RCONV
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEESX( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA], NOUT,
*       W, Z, [LDZ], RCONE, RCONV, [WORK], [LDWORK], [WORK2], [BWORK3],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL :: SELECT
 LOGICAL, DIMENSION(:) :: BWORK3
 REAL(8) :: RCONE, RCONV
```

```
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GEESX_64( JOBZ, SORTEV, SELECT, SENSE, [N], A, [LDA],
*       NOUT, W, Z, [LDZ], RCONE, RCONV, [WORK], [LDWORK], [WORK2],
*       [BWORK3], [INFO])
 CHARACTER(LEN=1) :: JOBZ, SORTEV, SENSE
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, NOUT, LDZ, LDWORK, INFO
 LOGICAL(8) :: SELECT
 LOGICAL(8), DIMENSION(:) :: BWORK3
 REAL(8) :: RCONE, RCONV
 REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgeesx(char jobz, char sortev, logical(*select)(COMPLEX*16), char sense, int n, doublecomplex *a, int lda, int *nout, doublecomplex *w, doublecomplex *z, int ldz, double *rcone, double *rconv, int *info);

void zgeesx_64(char jobz, char sortev, logical(*select)(COMPLEX*16), char sense, long n, doublecomplex *a, long lda, long *nout, doublecomplex *w, doublecomplex *z, long ldz, double *rcone, double *rconv, long *info);

PURPOSE

zgeesx computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues, the Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization A = Z*T*(Z**H).

Optionally, it also orders the eigenvalues on the diagonal of the Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (RCONDV). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see Section 4.10 of the LAPACK Users' Guide (where these quantities are called s and sep respectively).

A complex matrix is in Schur form if it is upper triangular.

ARGUMENTS

- **JOBZ (input)**

  = 'N': Schur vectors are not computed;

  = 'V': Schur vectors are computed.

- **SORTEV (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S': Eigenvalues are ordered (see SELECT).
```

- **SELECT (input)**
  SELECT must be declared EXTERNAL in the calling subroutine. If SORTEV = 'S', SELECT is used to select eigenvalues to order to the top left of the Schur form. If SORTEV = 'N', SELECT is not referenced. An eigenvalue `W(j)` is selected if `SELECT(W(j))` is true.
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
= 'E': Computed for average of selected eigenvalues only;

= 'V': Computed for selected right invariant subspace only;

= 'B': Computed for both.
If SENSE  = 'E', 'V' or 'B', SORTEV must equal 'S'.
```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A is overwritten by its Schur form T.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **NOUT (output)**
  If SORTEV = 'N', NOUT = 0. If SORTEV = 'S', NOUT = number of eigenvalues for which SELECT is true.
- **W (output)**
  W contains the computed eigenvalues, in the same order that they appear on the diagonal of the output Schur form T.
- **Z (output)**
  If JOBZ = 'V', Z contains the unitary matrix Z of Schur vectors. If JOBZ = 'N', Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.
- **RCONE (output)**
  If SENSE = 'E' or 'B', RCONE contains the reciprocal condition number for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONV (output)**
  If SENSE = 'V' or 'B', RCONV contains the reciprocal condition number for the selected right invariant subspace. Not referenced if SENSE = 'N' or 'E'.
- **WORK (workspace)**
  `dimension(LDWORK)` On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). Also, if SENSE = 'E' or 'V' or 'B', LDWORK >= 2*NOUT*(N-NOUT), where NOUT is the number of selected eigenvalues computed by this routine. Note that 2*NOUT*(N-NOUT) <= N*N/2. For good performance, LDWORK must generally be larger.
- **WORK2 (workspace)**
  `dimension(N)`
- **BWORK3 (workspace)**
  `dimension(N)` Not referenced if SORTEV = 'N'.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value.
```

> 0: if INFO = i, and i is

< = N: the QR algorithm failed to compute all the

eigenvalues; elements 1:ILO-1 and i+1:N of W contain those eigenvalues which have converged; if JOBZ = 'V', Z contains the transformation which reduces A to its partially converged Schur form. = N+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); = N+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy SELECT =.TRUE. This could also be caused by underflow due to scaling.

NAME

zgeev - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE ZGEEV( JOBVL, JOBVR, N, A, LDA, W, VL, LDVL, VR, LDVR,
*      WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGEEV_64( JOBVL, JOBVR, N, A, LDA, W, VL, LDVL, VR, LDVR,
*      WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEV( JOBVL, JOBVR, [N], A, [LDA], W, VL, [LDVL], VR,
*       [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, VL, VR
 INTEGER :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GEEV_64( JOBVL, JOBVR, [N], A, [LDA], W, VL, [LDVL], VR,
*       [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, VL, VR
 INTEGER(8) :: N, LDA, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgeev(char jobvl, char jobvr, int n, doublecomplex *a, int lda, doublecomplex *w, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int *info);

void zgeev_64(char jobvl, char jobvr, long n, doublecomplex *a, long lda, doublecomplex *w, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long *info);

PURPOSE

zgeev computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N': left eigenvectors of A are not computed;

  = 'V': left eigenvectors of are computed.
  ```
- **JOBVR (input)**

  ```
  = 'N': right eigenvectors of A are not computed;

  = 'V': right eigenvectors of A are computed.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **W (output)**
  W contains the computed eigenvalues.
- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. `u(j)` = VL(:,j), the j-th column of VL.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. `v(j)` = VR(:,j), the j-th column of VR.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors have been computed;
elements and i+1:N of W contain eigenvalues which have
converged.
```

NAME

zgeevx - compute for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors

---

SYNOPSIS

```
 SUBROUTINE ZGEEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, W, VL,
*      LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 DOUBLE COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 DOUBLE PRECISION ABNRM
 DOUBLE PRECISION SCALE(*), RCONE(*), RCONV(*), WORK2(*)

 SUBROUTINE ZGEEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, W, VL,
*      LDVL, VR, LDVR, ILO, IHI, SCALE, ABNRM, RCONE, RCONV, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 DOUBLE COMPLEX A(LDA,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 DOUBLE PRECISION ABNRM
 DOUBLE PRECISION SCALE(*), RCONE(*), RCONV(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], W, VL,
*       [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV, [WORK],
*       [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, VL, VR
 INTEGER :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL(8) :: ABNRM
 REAL(8), DIMENSION(:) :: SCALE, RCONE, RCONV, WORK2

 SUBROUTINE GEEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], W,
*       VL, [LDVL], VR, [LDVR], ILO, IHI, SCALE, ABNRM, RCONE, RCONV,
```

```
*       [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, VL, VR
 INTEGER(8) :: N, LDA, LDVL, LDVR, ILO, IHI, LDWORK, INFO
 REAL(8) :: ABNRM
 REAL(8), DIMENSION(:) :: SCALE, RCONE, RCONV, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgeevx(char balanc, char jobvl, char jobvr, char sense, int n, doublecomplex *a, int lda, doublecomplex *w, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int *ilo, int *ihi, double *scale, double *abnrm, double *rcone, double *rconv, int *info);

void zgeevx_64(char balanc, char jobvl, char jobvr, char sense, long n, doublecomplex *a, long lda, doublecomplex *w, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long *ilo, long *ihi, double *scale, double *abnrm, double *rcone, double *rconv, long *info);

PURPOSE

zgeevx computes for an N-by-N complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, SCALE, and ABNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right

eigenvectors (RCONDV).

The right eigenvector `v(j)` of A satisfies

```
A * v(j) = lambda(j) * v(j)
```

where `lambda(j)` is its eigenvalue.

The left eigenvector `u(j)` of A satisfies

```
u(j)**H * A = lambda(j) * u(j)**H
```

where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D**(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide.

ARGUMENTS

- **BALANC (input)**
  Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. = 'N': Do not diagonally scale or permute;

```
 = 'P': Perform permutations to make the matrix more nearly
upper triangular. Do not diagonally scale;
 = 'S': Diagonally scale the matrix, ie. replace A by
D*A*D**(-1), where D is a diagonal matrix chosen
to make the rows and columns of A more equal in
norm. Do not permute;
 = 'B': Both diagonally scale and permute A.
```

  Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

- **JOBVL (input)**

```
 = 'N': left eigenvectors of A are not computed;

 = 'V': left eigenvectors of A are computed.
If SENSE  = 'E' or 'B', JOBVL must  = 'V'.
```

- **JOBVR (input)**

```
 = 'N': right eigenvectors of A are not computed;

 = 'V': right eigenvectors of A are computed.
If SENSE  = 'E' or 'B', JOBVR must  = 'V'.
```

- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': None are computed;

```
 = 'E': Computed for eigenvalues only;

 = 'V': Computed for right eigenvectors only;

 = 'B': Computed for eigenvalues and right eigenvectors.
```

  If SENSE = 'E' or 'B', both left and right eigenvectors must also be computed (JOBVL = 'V' and JOBVR = 'V').

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the N-by-N matrix A. On exit, A has been overwritten. If JOBVL = 'V' or JOBVR = 'V', A contains the Schur form of the balanced version of the matrix A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  W contains the computed eigenvalues.

- **VL (output)**
  If JOBVL = 'V', the left eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. If JOBVL = 'N', VL is not referenced. `u(j)` = VL(:,j), the j-th column of VL.

- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= 1; if JOBVL = 'V', LDVL >= N.

- **VR (output)**
  If JOBVR = 'V', the right eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. If JOBVR = 'N', VR is not referenced. `v(j)` = VR(:,j), the j-th column of VR.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= 1; if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced `A(i,j)` = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **IHI (output)**
  ILO and IHI are integer values determined when A was balanced. The balanced `A(i,j)` = 0 if I > J and J = 1,...,ILO-1 or I = IHI+1,...,N.
- **SCALE (output)**
  Details of the permutations and scaling factors applied when balancing A. If `P(j)` is the index of the row and column interchanged with row and column j, and `D(j)` is the scaling factor applied to row and column j, then `SCALE(J)` = P(J), for J = 1,...,ILO-1 = D(J), for J = ILO,...,IHI = `P(J)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
- **RCONE (output)**
  `RCONE(j)` is the reciprocal condition number of the j-th eigenvalue.
- **RCONV (output)**
  `RCONV(j)` is the reciprocal condition number of the j-th right eigenvector.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. If SENSE = 'N' or 'E', LDWORK >= max(1,2*N), and if SENSE = 'V' or 'B', LDWORK >= N*N+2*N. For good performance, LDWORK must generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the QR algorithm failed to compute all the
eigenvalues, and no eigenvectors or condition numbers
have been computed; elements 1:ILO-1 and i+1:N of W
contain eigenvalues which have converged.
```

NAME

zgegs - routine is deprecated and has been replaced by routine CGGES

SYNOPSIS

```
 SUBROUTINE ZGEGS( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGEGS_64( JOBVSL, JOBVSR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VSL, LDVSL, VSR, LDVSR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGS( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GEGS_64( JOBVSL, JOBVSR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK], [LDWORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER(8) :: N, LDA, LDB, LDVSL, LDVSR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgegs(char jobvsl, char jobvsr, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, int ldvsl, doublecomplex *vsr, int ldvsr, int *info);

void zgegs_64(char jobvsl, char jobvsr, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, long ldvsl, doublecomplex *vsr, long ldvsr, long *info);

PURPOSE

zgegs routine is deprecated and has been replaced by routine CGGES.

CGEGS computes for a pair of N-by-N complex nonsymmetric matrices A, B: the generalized eigenvalues (alpha, beta), the complex Schur form (A, B), and optionally left and/or right Schur vectors (VSL and VSR).

(If only the generalized eigenvalues are needed, use the driver CGEGV instead.)

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations", by G. Golub & C. van Loan (Johns Hopkins U. Press)

The (generalized) Schur form of a pair of matrices is the result of multiplying both matrices on the left by one unitary matrix and both on the right by another unitary matrix, these two unitary matrices being chosen so as to bring the pair of matrices into upper triangular form with the diagonal elements of B being non-negative real numbers (this is also called complex Schur form.)

The left and right Schur vectors are the columns of VSL and VSR, respectively, where VSL and VSR are the unitary matrices

which reduce A and B to Schur form:

Schur form of (A,B) = ((VSL)**H A (VSR), (VSL)**H B (VSR))

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of A.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**

On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) Schur vectors are to be computed. On exit, the generalized Schur form of B.

- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHA(j), j =1,...,N and BETA(j), j =1,...,N are the diagonals of the complex Schur form (A,B) output by CGEGS. The `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See the description of ALPHA.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL >= 1, and if JOBVSL = 'V', LDVSL >= N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. (See ``Purpose'', above.) Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR >= 1, and if JOBVSR = 'V', LDVSR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,2*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for CGEQRF, CUNMQR, and CUNGQR.) Then compute: NB as the MAX of the blocksizes for CGEQRF, CUNMQR, and CUNGQR; the optimal LDWORK is N*(NB+1).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from CGGBAL

 =N+2: error return from CGEQRF

 =N+3: error return from CUNMQR

 =N+4: error return from CUNGQR

 =N+5: error return from CGGHRD

 =N+6: error return from CHGEQZ (other than failed
iteration)
 =N+7: error return from CGGBAK (computing VSL)
```

=N+8: error return from CGGBAK (computing VSR)

=N+9: error return from CLASCL (various places)

NAME

zgegv - routine is deprecated and has been replaced by routine CGGEV

SYNOPSIS

```
 SUBROUTINE ZGEGV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA, VL,
*      LDVL, VR, LDVR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGEGV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VL, LDVL, VR, LDVR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GEGV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA, BETA,
*       VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GEGV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VL, [LDVL], VR, [LDVR], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LDWORK, INFO
 REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgegv(char jobvl, char jobvr, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int *info);

void zgegv_64(char jobvl, char jobvr, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long *info);

PURPOSE

zgegv routine is deprecated and has been replaced by routine CGGEV.

CGEGV computes for a pair of N-by-N complex nonsymmetric matrices A and B, the generalized eigenvalues (alpha, beta), and optionally, the left and/or right generalized eigenvectors (VL and VR).

A generalized eigenvalue for a pair of matrices (A,B) is, roughly speaking, a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero. A good beginning reference is the book, ``Matrix Computations", by G. Golub & C. van Loan (Johns Hopkins U. Press)

A right generalized eigenvector corresponding to a generalized eigenvalue w for a pair of matrices (A,B) is a vector r such that (A - w B) r = 0 . A left generalized eigenvector is a vector l such that l**H * (A - w B) = 0, where l**H is the

conjugate-transpose of l.

Note: this routine performs ``full balancing" on A and B. See ``Further Details", below.

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.

- **A (input/output)**
  On entry, the first of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of A on exit, see ``Further Details", below.)

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**

On entry, the second of the pair of matrices whose generalized eigenvalues and (optionally) generalized eigenvectors are to be computed. On exit, the contents will have been destroyed. (For a description of the contents of B on exit, see ``Further Details'', below.)

- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/VL(j), j =1,...,N, will be the generalized eigenvalues.

  Note: the quotients `ALPHA(j)/VL(j)` may easily over- or underflow, and `VL(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and VL always less than and usually comparable with norm(B).
- **BETA (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL > = 1, and if JOBVL = 'V', LDVL > = N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors. (See ``Purpose'', above.) Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1, *except* that for eigenvalues with alpha =beta =0, a zero vector will be returned as the corresponding eigenvector. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR > = 1, and if JOBVR = 'V', LDVR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,2*N). For good performance, LDWORK must generally be larger. To compute the optimal value of LDWORK, call ILAENV to get blocksizes (for CGEQRF, CUNMQR, and CUNGQR.) Then compute: NB as the MAX of the blocksizes for CGEQRF, CUNMQR, and CUNGQR; The optimal LDWORK is MAX(2*N, N*(NB+1)).

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and VL(j) should be
correct for j =INFO+1,...,N.
 > N:  errors that usually indicate LAPACK problems:

 =N+1: error return from CGGBAL

 =N+2: error return from CGEQRF
```

```
 =N+3: error return from CUNMQR

 =N+4: error return from CUNGQR

 =N+5: error return from CGGHRD

 =N+6: error return from CHGEQZ (other than failed
iteration)
 =N+7: error return from CTGEVC

 =N+8: error return from CGGBAK (computing VL)

 =N+9: error return from CGGBAK (computing VR)

 =N+10: error return from CLASCL (various calls)
```

FURTHER DETAILS

Balancing

This driver calls CGGBAL to both permute and scale rows and columns of A and B. The permutations PL and PR are chosen so that PL*A*PR and PL*B*R will be upper triangular except for the diagonal blocks `A(i:j,i:j)` and B(i:j,i:j), with i and j as close together as possible. The diagonal scaling matrices DL and DR are chosen so that the pair DL*PL*A*PR*DR, DL*PL*B*PR*DR have elements close to one (except for the elements that start out zero.)

After the eigenvalues and eigenvectors of the balanced matrices have been computed, CGGBAK transforms the eigenvectors back to what they would have been (in perfect arithmetic) if they had not been balanced.

Contents of A and B on Exit

-------- -- - --- - -- ----

If any eigenvectors are computed (either JOBVL ='V' or JOBVR ='V' or both), then on exit the arrays A and B will contain the complex Schur form[*] of the ``balanced'' versions of A and B. If no eigenvectors are computed, then only the diagonal blocks will be correct.

[*] In other words, upper triangular form.

NAME

zgehrd - reduce a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation

---

SYNOPSIS

```
  SUBROUTINE ZGEHRD( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN, INFO)
  DOUBLE COMPLEX A(LDA,*), TAU(*), WORKIN(*)
  INTEGER N, ILO, IHI, LDA, LWORKIN, INFO

  SUBROUTINE ZGEHRD_64( N, ILO, IHI, A, LDA, TAU, WORKIN, LWORKIN,
 *      INFO)
  DOUBLE COMPLEX A(LDA,*), TAU(*), WORKIN(*)
  INTEGER*8 N, ILO, IHI, LDA, LWORKIN, INFO
```

F95 INTERFACE

```
  SUBROUTINE GEHRD( [N], ILO, IHI, A, [LDA], TAU, [WORKIN], [LWORKIN],
 *       [INFO])
  COMPLEX(8), DIMENSION(:) :: TAU, WORKIN
  COMPLEX(8), DIMENSION(:,:) :: A
  INTEGER :: N, ILO, IHI, LDA, LWORKIN, INFO

  SUBROUTINE GEHRD_64( [N], ILO, IHI, A, [LDA], TAU, [WORKIN],
 *       [LWORKIN], [INFO])
  COMPLEX(8), DIMENSION(:) :: TAU, WORKIN
  COMPLEX(8), DIMENSION(:,:) :: A
  INTEGER(8) :: N, ILO, IHI, LDA, LWORKIN, INFO
```

C INTERFACE

#include <sunperf.h>

void zgehrd(int n, int ilo, int ihi, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zgehrd_64(long n, long ilo, long ihi, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zgehrd reduces a complex general matrix A to upper Hessenberg form H by a unitary similarity transformation: Q' * A * Q = H .

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGEBAL; otherwise they should be set to 1 and N respectively. See Further Details.
- **IHI (input)**
  See the description of ILO.
- **A (input/output)**
  On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details). Elements 1:ILO-1 and IHI:N-1 of TAU are set to zero.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The length of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of (ihi-ilo) elementary reflectors

```
Q  = H(ilo) H(ilo+1) . . . H(ihi-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0, `v(i+1)` = 1 and `v(ihi+1:n)` = 0; `v(i+2:ihi)` is stored on exit in A(i+2:ihi,i), and tau in TAU(i).

The contents of A are illustrated by the following example, with n = 7, ilo = 2 and ihi = 6:

on entry, on exit,

(a a a a a a a) (a a h h h h a) (a a a a a a) (a h h h h a) (a a a a a a) (h h h h h h) (a a a a a a) (v2 h h h h h) (a a a a a a) (v2 v3 h h h h) (a a a a a a) (v2 v3 v4 h h h) (a) (a)

where a denotes an element of the original matrix A, h denotes a modified element of the upper Hessenberg matrix H, and vi denotes an element of the vector defining H(i).

NAME

zgelqf - compute an LQ factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE ZGELQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE ZGELQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GELQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GELQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgelqf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zgelqf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zgelqf computes an LQ factorization of a complex M-by-N matrix A: A = L * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and below the diagonal of the array contain the m-by-min(m,n) lower trapezoidal matrix L (L is lower triangular if m <= n); the elements above the diagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k)' . . . H(2)' H(1)', where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `conjg(v(i+1:n))` is stored

on exit in A(i,i+1:n), and tau in TAU(i).

NAME

zgels - solve overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A

SYNOPSIS

```
 SUBROUTINE ZGELS( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK, LDWORK,
*      INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, LDWORK, INFO

 SUBROUTINE ZGELS_64( TRANSA, M, N, NRHS, A, LDA, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GELS( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, LDWORK, INFO

 SUBROUTINE GELS_64( [TRANSA], [M], [N], [NRHS], A, [LDA], B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgels(char transa, int m, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zgels_64(char transa, long m, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zgels solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||.

2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B.

3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B.

4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

ARGUMENTS

- **TRANSA (input)**

  ```
  = 'N': the linear system involves A;

  = 'C': the linear system involves A**H.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. if M > = N, A is overwritten by details of its QR factorization as returned by CGEQRF; if M < N, A is overwritten by details of its LQ factorization as returned by CGELQF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the matrix B of right hand side vectors, stored columnwise; B is M-by-NRHS if TRANSA = 'N', or

N-by-NRHS if TRANSA = 'C'. On exit, B is overwritten by the solution vectors, stored columnwise: if TRANSA = 'N' and m >= n, rows 1 to n of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements N+1 to M in that column; if TRANSA = 'N' and m < n, rows 1 to N of B contain the minimum norm solution vectors; if TRANSA = 'C' and m >= n, rows 1 to M of B contain the minimum norm solution vectors; if TRANSA = 'C' and m < n, rows 1 to M of B contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements M+1 to N in that column.

- **LDB (input)**
  The leading dimension of the array B. LDB >= MAX(1,M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1, MN + max(MN, NRHS)). For optimal performance, LDWORK >= max(1, MN + max(MN, NRHS)*NB). where MN = `min(M,N)` and NB is the optimum block size.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgelsd - compute the minimum-norm solution to a real linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE ZGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK,
*      LWORK, RWORK, IWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION S(*), RWORK(*)

 SUBROUTINE ZGELSD_64( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK,
*      WORK, LWORK, RWORK, IWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION S(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSD( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: S, RWORK

 SUBROUTINE GELSD_64( [M], [N], [NRHS], A, [LDA], B, [LDB], S, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: S, RWORK
```

C INTERFACE

#include <sunperf.h>

void zgelsd(int m, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, double *s, double rcond, int *rank, int *info);

void zgelsd_64(long m, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, double *s, double rcond, long *rank, long *info);

PURPOSE

zgelsd computes the minimum-norm solution to a real linear least squares problem: minimize 2-norm(| b - A*x |)

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The problem is solved in three steps:

(1) Reduce the coefficient matrix A to bidiagonal form with Householder tranformations, reducing the original problem into a ``bidiagonal least squares problem'' (BLS)

(2) Solve the BLS using a divide and conquer approach.

(3) Apply back all the Householder tranformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and RANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,M,N).
- **S (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = S(1)/S(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `S(i)` < = RCOND*S(1) are treated as zero. If RCOND < 0, machine precision is used instead.

- **RANK (output)**
  The effective rank of A, i.e., the number of singular values which are greater than RCOND*S(1).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. The exact minimum amount of workspace needed depends on M, N and NRHS. If M > = N, LWORK > = 2*N + N*NRHS. If M < N, LWORK > = 2*M + M*NRHS. For good performance, LWORK should generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  If M > = N, LRWORK > = 8*N + 2*N*SMLSIZ + 8*N*NLVL + N*NRHS. If M < N, LRWORK > = 8*M + 2*M*SMLSIZ + 8*M*NLVL + M*NRHS. SMLSIZ is returned by ILAENV and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and NLVL = INT(LOG_2(MIN(M,N)/(SMLSIZ+1))) + 1
- **IWORK (workspace)**
  LIWORK > = 3 * MINMN * NLVL + 11 * MINMN, where MINMN = MIN(M,N).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Ming Gu and Ren-Cang Li, Computer Science Division, University of California at Berkeley, USA

Osni Marques, LBNL/NERSC, USA
```

NAME

zgelss - compute the minimum norm solution to a complex linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE ZGELSS( M, N, NRHS, A, LDA, B, LDB, SING, RCOND, IRANK,
*      WORK, LDWORK, WORK2, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SING(*), WORK2(*)

 SUBROUTINE ZGELSS_64( M, N, NRHS, A, LDA, B, LDB, SING, RCOND,
*      IRANK, WORK, LDWORK, WORK2, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SING(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSS( [M], [N], [NRHS], A, [LDA], B, [LDB], SING, RCOND,
*       IRANK, [WORK], [LDWORK], [WORK2], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SING, WORK2

 SUBROUTINE GELSS_64( [M], [N], [NRHS], A, [LDA], B, [LDB], SING,
*       RCOND, IRANK, [WORK], [LDWORK], [WORK2], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, LDWORK, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SING, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgelss(int m, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, double *sing, double rcond, int *irank, int *info);

void zgelss_64(long m, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, double *sing, double rcond, long *irank, long *info);

PURPOSE

zgelss computes the minimum norm solution to a complex linear least squares problem:

Minimize 2-norm(| b - A*x |).

using the singular value decomposition (SVD) of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The effective rank of A is determined by treating as zero those singular values which are less than RCOND times the largest singular value.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the first `min(m,n)` rows of A are overwritten with its right singular vectors, stored rowwise.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, B is overwritten by the N-by-NRHS solution matrix X. If m > = n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements n+1:m in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,M,N).
- **SING (output)**
  The singular values of A in decreasing order. The condition number of A in the 2-norm = SING(1)/SING(min(m,n)).
- **RCOND (input)**
  RCOND is used to determine the effective rank of A. Singular values `SING(i)` < = RCOND*SING(1) are treated as zero. If RCOND < 0, machine precision is used instead.

- **IRANK (output)**
  The effective rank of A, i.e., the number of singular values which are greater than RCOND*SING(1).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1, and also: LDWORK >= 2*min(M,N) + `max(M,N,NRHS)`
  For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(5*min(M,N))`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  the algorithm for computing the SVD failed to converge;
if INFO  = i, i off-diagonal elements of an intermediate
bidiagonal form did not converge to zero.
```

NAME

zgelsx - routine is deprecated and has been replaced by routine CGELSY

SYNOPSIS

```
 SUBROUTINE ZGELSX( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND, IRANK,
*      WORK, WORK2, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER JPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZGELSX_64( M, N, NRHS, A, LDA, B, LDB, JPIVOT, RCOND,
*      IRANK, WORK, WORK2, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER*8 JPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSX( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [WORK2], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER, DIMENSION(:) :: JPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE GELSX_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPIVOT,
*       RCOND, IRANK, [WORK], [WORK2], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, NRHS, LDA, LDB, IRANK, INFO
 INTEGER(8), DIMENSION(:) :: JPIVOT
```

```
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgelsx(int m, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *jpivot, double rcond, int *irank, int *info);

void zgelsx_64(long m, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *jpivot, double rcond, long *irank, long *info);

PURPOSE

zgelsx routine is deprecated and has been replaced by routine CGELSY.

CGELSX computes the minimum-norm solution to a complex linear least squares problem:

```
    minimize || A * X - B ||
```

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by unitary transformations from the right, arriving at the complete orthogonal factorization:

```
   A * P = Q * [ T11 0 ] * Z

               [  0  0 ]
```

The minimum-norm solution is then

```
   X = P * Z' [ inv(T11)*Q1'*B ]

              [        0       ]
```

where Q1 consists of the first RANK columns of Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X. If m >= n and IRANK = n, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements N+1:M in that column.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is an initial column, otherwise it is a free column. Before the QR factorization of A, all initial columns are permuted to the leading positions; only the remaining free columns are moved as a result of column pivoting during the factorization. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **IRANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  (min(M,N) + max(N, 2*min(M,N)+NRHS)),
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgelsy - compute the minimum-norm solution to a complex linear least squares problem

---

SYNOPSIS

```
 SUBROUTINE ZGELSY( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, RWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER JPVT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZGELSY_64( M, N, NRHS, A, LDA, B, LDB, JPVT, RCOND, RANK,
*      WORK, LWORK, RWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER*8 JPVT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GELSY( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT, RCOND,
*       RANK, [WORK], [LWORK], [RWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE GELSY_64( [M], [N], [NRHS], A, [LDA], B, [LDB], JPVT,
*       RCOND, RANK, [WORK], [LWORK], [RWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
```

```
INTEGER(8) :: M, N, NRHS, LDA, LDB, RANK, LWORK, INFO
INTEGER(8), DIMENSION(:) :: JPVT
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zgelsy(int m, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *jpvt, double rcond, int *rank, int *info);

void zgelsy_64(long m, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *jpvt, double rcond, long *rank, long *info);

PURPOSE

zgelsy computes the minimum-norm solution to a complex linear least squares problem: minimize || A * X - B ||

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

The routine first computes a QR factorization with column pivoting: A * P = Q * [R11 R12]

```
                [  0  R22 ]
```

with R11 defined as the largest leading submatrix whose estimated condition number is less than 1/RCOND. The order of R11, RANK, is the effective rank of A.

Then, R22 is considered to be negligible, and R12 is annihilated by unitary transformations from the right, arriving at the complete orthogonal factorization:

```
A * P = Q * [ T11 0 ] * Z

            [  0  0 ]
```

The minimum-norm solution is then

```
X = P * Z' [ inv(T11)*Q1'*B ]

           [        0       ]
```

where Q1 consists of the first RANK columns of Q.

This routine is basically identical to the original xGELSX except three differences:

```
o The permutation of matrix B (the right hand side) is faster and
  more simple.

o The call to the subroutine xGEQPF has been substituted by the
  the call to the subroutine xGEQP3. This subroutine is a Blas-3
```

```
  version of the QR factorization with column pivoting.
o Matrix B (the right hand side) is updated with Blas-3.
```

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of matrices B and X. NRHS >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A has been overwritten by details of its complete orthogonal factorization.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the M-by-NRHS right hand side matrix B. On exit, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,M,N).
- **JPVT (input/output)**
  On entry, if `JPVT(i)` .ne. 0, the i-th column of A is permuted to the front of AP, otherwise column i is a free column. On exit, if `JPVT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **RCOND (input)**
  RCOND is used to determine the effective rank of A, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of A, whose estimated condition number < 1/RCOND.
- **RANK (output)**
  The effective rank of A, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of A.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. The unblocked strategy requires that: LWORK >= MN + MAX(2*MN, N+1, MN+NRHS) where MN = min(M,N). The block algorithm requires that: LWORK >= MN + MAX(2*MN, NB*(N+1), MN+MN*NB, MN+NB*NRHS) where NB is an upper bound on the blocksize returned by ILAENV for the routines CGEQP3, CTZRZF, CTZRQF, CUNMQR, and CUNMRZ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
E. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain

---

NAME

zgemm - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C

SYNOPSIS

```
 SUBROUTINE ZGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, K, LDA, LDB, LDC

 SUBROUTINE ZGEMM_64( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 TRANSA, TRANSB
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, K, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE GEMM( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A, [LDA],
*       B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, K, LDA, LDB, LDC

 SUBROUTINE GEMM_64( [TRANSA], [TRANSB], [M], [N], [K], ALPHA, A,
*       [LDA], B, [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: TRANSA, TRANSB
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, K, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void zgemm(char transa, char transb, int m, int n, int k, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex beta, doublecomplex *c, int ldc);

void zgemm_64(char transa, char transb, long m, long n, long k, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex beta, doublecomplex *c, long ldc);

PURPOSE

zgemm performs one of the matrix-matrix operations

C := alpha*op(A)*op(B) + beta*C

where op(X) is one of

`op(X)` = X or `op(X)` = X' or `op(X)` = conjg(X'), alpha and beta are scalars, and A, B and C are matrices, with `op(A)` an m by k matrix, `op(B)` a k by n matrix and C an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n', op(A) = A.

  TRANSA = 'T' or 't', op(A) = A'.

  TRANSA = 'C' or 'c', op(A) = conjg(A').

  Unchanged on exit.
- **TRANSB (input)**
  On entry, TRANSB specifies the form of op(B) to be used in the matrix multiplication as follows:

  TRANSB = 'N' or 'n', op(B) = B.

  TRANSB = 'T' or 't', op(B) = B'.

  TRANSB = 'C' or 'c', op(B) = conjg(B').

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix op(A) and of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **K (input)**
  On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K

> = 0. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  K when TRANSA = 'N' or 'n', and is M otherwise. Before entry with TRANSA = 'N' or 'n', the leading M by K part of the array A must contain the matrix A, otherwise the leading K by M part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA > = max(1, M), otherwise LDA > = max(1, K). Unchanged on exit.
- **B (input)**
  n when TRANSB = 'N' or 'n', and is k otherwise. Before entry with TRANSB = 'N' or 'n', the leading k by n part of the array B must contain the matrix B, otherwise the leading n by k part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSB = 'N' or 'n' then LDB > = max(1, k), otherwise LDB > = max(1, n). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n matrix (alpha*op(A)*op(B) + beta*C).
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

zgemv - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE ZGEMV( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER M, N, LDA, INCX, INCY

 SUBROUTINE ZGEMV_64( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 M, N, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE GEMV( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, INCX, INCY

 SUBROUTINE GEMV_64( [TRANSA], [M], [N], ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zgemv(char transa, int m, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *x, int incx, doublecomplex beta, doublecomplex *y, int incy);

void zgemv_64(char transa, long m, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *x, long incx, doublecomplex beta, doublecomplex *y, long incy);

PURPOSE

zgemv performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y, or y := alpha*conjg(A')*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

ARGUMENTS

- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' y : = alpha*A*x + beta*y.

  TRANSA = 'T' or 't' y : = alpha*A'*x + beta*y.

  TRANSA = 'C' or 'c' y : = alpha*conjg(A')*x + beta*y.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)) when TRANSA = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (m - 1)*abs(INCY)) when TRANSA = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before

entry with BETA non-zero, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zgeqlf - compute a QL factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE ZGEQLF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE ZGEQLF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GEQLF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GEQLF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgeqlf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zgeqlf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zgeqlf computes a QL factorization of a complex M-by-N matrix A: A = Q * L.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m >= n, the lower triangle of the subarray `A(m-n+1:m,1:n)` contains the N-by-N lower triangular matrix L; if m <= n, the elements on and below the (n-m)-th superdiagonal contain the M-by-N lower trapezoidal matrix L; the remaining elements, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(k) . . . H(2) H(1), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(m-k+i+1:m)` = 0 and `v(m-k+i)` = 1; `v(1:m-k+i-1)` is stored on exit in A(1:m-k+i-1,n-k+i), and tau in TAU(i).

NAME

zgeqp3 - compute a QR factorization with column pivoting of a matrix A

SYNOPSIS

```
 SUBROUTINE ZGEQP3( M, N, A, LDA, JPVT, TAU, WORK, LWORK, RWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
 INTEGER M, N, LDA, LWORK, INFO
 INTEGER JPVT(*)
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZGEQP3_64( M, N, A, LDA, JPVT, TAU, WORK, LWORK, RWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
 INTEGER*8 M, N, LDA, LWORK, INFO
 INTEGER*8 JPVT(*)
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GEQP3( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [RWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: M, N, LDA, LWORK, INFO
 INTEGER, DIMENSION(:) :: JPVT
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE GEQP3_64( [M], [N], A, [LDA], JPVT, TAU, [WORK], [LWORK],
*       [RWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: M, N, LDA, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: JPVT
 REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zgeqp3(int m, int n, doublecomplex *a, int lda, int *jpvt, doublecomplex *tau, int *info);

void zgeqp3_64(long m, long n, doublecomplex *a, long lda, long *jpvt, doublecomplex *tau, long *info);

PURPOSE

zgeqp3 computes a QR factorization with column pivoting of a matrix A: A*P = Q*R using Level 3 BLAS.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper trapezoidal matrix R; the elements below the diagonal, together with the array TAU, represent the unitary matrix Q as a product of `min(M,N)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **JPVT (input/output)**
  On entry, if JPVT(J).ne.0, the J-th column of A is permuted to the front of A*P (a leading column); if `JPVT(J)` =0, the J-th column of A is a free column. On exit, if `JPVT(J)` =K, then the J-th column of A*P was the the K-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = N+1. For optimal performance LWORK > = (N+1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  `= 0: successful exit.`

```
< 0: if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a real/complex scalar, and v is a real/complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

Based on contributions by

```
G. Quintana-Orti, Depto. de Informatica, Universidad Jaime I, Spain
X. Sun, Computer Science Dept., Duke University, USA
```

NAME

zgeqpf - routine is deprecated and has been replaced by routine CGEQP3

SYNOPSIS

```
SUBROUTINE ZGEQPF( M, N, A, LDA, JPIVOT, TAU, WORK, WORK2, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, INFO
INTEGER JPIVOT(*)
DOUBLE PRECISION WORK2(*)

SUBROUTINE ZGEQPF_64( M, N, A, LDA, JPIVOT, TAU, WORK, WORK2, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 JPIVOT(*)
DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE GEQPF( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [WORK2],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: JPIVOT
REAL(8), DIMENSION(:) :: WORK2

SUBROUTINE GEQPF_64( [M], [N], A, [LDA], JPIVOT, TAU, [WORK], [WORK2],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: JPIVOT
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zgeqpf(int m, int n, doublecomplex *a, int lda, int *jpivot, doublecomplex *tau, int *info);

void zgeqpf_64(long m, long n, doublecomplex *a, long lda, long *jpivot, doublecomplex *tau, long *info);

PURPOSE

zgeqpf routine is deprecated and has been replaced by routine CGEQP3.

CGEQPF computes a QR factorization with column pivoting of a complex M-by-N matrix A: A*P = Q*R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the upper triangle of the array contains the min(M,N)-by-N upper triangular matrix R; the elements below the diagonal, together with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **JPIVOT (input)**
  On entry, if `JPIVOT(i)` .ne. 0, the i-th column of A is permuted to the front of A*P (a leading column); if `JPIVOT(i)` = 0, the i-th column of A is a free column. On exit, if `JPIVOT(i)` = k, then the i-th column of A*P was the k-th column of A.
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n)
```

Each `H(i)` has the form

```
H  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i).

The matrix P is represented in jpvt as follows: If

```
jpvt(j)  = i
```

then the jth column of P is the ith canonical unit vector.

NAME

zgeqrf - compute a QR factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE ZGEQRF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE ZGEQRF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GEQRF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GEQRF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgeqrf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zgeqrf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zgeqrf computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, the elements on and above the diagonal of the array contain the min(M,N)-by-N upper trapezoidal matrix R (R is upper triangular if m >= n); the elements below the diagonal, with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N). For optimum performance LDWORK >= N*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:m)` is stored on exit in A(i+1:m,i), and tau in TAU(i).

NAME

zgerc - perform the rank 1 operation A := alpha*x*conjg(y') + A

SYNOPSIS

```
SUBROUTINE ZGERC( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER M, N, INCX, INCY, LDA

SUBROUTINE ZGERC_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 M, N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE GERC( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, INCX, INCY, LDA

SUBROUTINE GERC_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void zgerc(int m, int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy, doublecomplex *a, int lda);

void zgerc_64(long m, long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy, doublecomplex *a, long lda);

PURPOSE

zgerc performs the rank 1 operation A := alpha*x*conjg(y') + A where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

zgerfs - improve the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZGERFS( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGERFS_64( TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GERFS( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GERFS_64( [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgerfs(char transa, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zgerfs_64(char transa, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zgerfs improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The original N-by-N matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CGETRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgerqf - compute an RQ factorization of a complex M-by-N matrix A

SYNOPSIS

```
SUBROUTINE ZGERQF( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LDWORK, INFO

SUBROUTINE ZGERQF_64( M, N, A, LDA, TAU, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LDWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE GERQF( [M], [N], A, [LDA], TAU, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LDWORK, INFO

SUBROUTINE GERQF_64( [M], [N], A, [LDA], TAU, [WORK], [LDWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgerqf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zgerqf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zgerqf computes an RQ factorization of a complex M-by-N matrix A: A = R * Q.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if m <= n, the upper triangle of the subarray `A(1:m,n-m+1:n)` contains the M-by-M upper triangular matrix R; if m >= n, the elements on and above the (m-n)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAU, represent the unitary matrix Q as a product of `min(m,n)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,M). For optimum performance LDWORK >= M*NB, where NB is the optimal blocksize.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1)' H(2)' . . . H(k)', where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `conjg(v(1:n-k+i-1))` is stored on exit in A(m-k+i,1:n-k+i-1), and tau in TAU(i).

NAME

zgeru - perform the rank 1 operation A := alpha*x*y' + A

SYNOPSIS

```
SUBROUTINE ZGERU( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER M, N, INCX, INCY, LDA

SUBROUTINE ZGERU_64( M, N, ALPHA, X, INCX, Y, INCY, A, LDA)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 M, N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE GER( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, INCX, INCY, LDA

SUBROUTINE GER_64( [M], [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void zgeru(int m, int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy, doublecomplex *a, int lda);

void zgeru_64(long m, long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy, doublecomplex *a, long lda);

PURPOSE

zgeru performs the rank 1 operation A := alpha*x*y' + A where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

ARGUMENTS

- **M (input)**
  On entry, M specifies the number of rows of the matrix A. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (m - 1)*abs(INCX)). Before entry, the incremented array X must contain the m element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.
- **A (input/output)**
  Before entry, the leading m by n part of the array A must contain the matrix of coefficients. On exit, A is overwritten by the updated matrix.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, m). Unchanged on exit.

NAME

zgesdd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method

---

SYNOPSIS

```
 SUBROUTINE ZGESDD( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, RWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 DOUBLE COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION S(*), RWORK(*)

 SUBROUTINE ZGESDD_64( JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK,
*      LWORK, RWORK, IWORK, INFO)
 CHARACTER * 1 JOBZ
 DOUBLE COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER*8 M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION S(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GESDD( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT, [LDVT],
*       [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, U, VT
 INTEGER :: M, N, LDA, LDU, LDVT, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: S, RWORK

 SUBROUTINE GESDD_64( JOBZ, [M], [N], A, [LDA], S, U, [LDU], VT,
*       [LDVT], [WORK], [LWORK], [RWORK], [IWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ
```

```
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, U, VT
INTEGER(8) :: M, N, LDA, LDU, LDVT, LWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: S, RWORK
```

C INTERFACE

#include <sunperf.h>

void zgesdd(char jobz, int m, int n, doublecomplex *a, int lda, double *s, doublecomplex *u, int ldu, doublecomplex *vt, int ldvt, int *info);

void zgesdd_64(char jobz, long m, long n, doublecomplex *a, long lda, double *s, doublecomplex *u, long ldu, doublecomplex *vt, long ldvt, long *info);

PURPOSE

zgesdd computes the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors, by using divide-and-conquer method. The SVD is written = U * SIGMA * conjugate-transpose(V)

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M unitary matrix, and V is an N-by-N unitary matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns VT = V**H, not V.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**
  Specifies options for computing all or part of the matrix U:

```
 = 'A':  all M columns of U and all N rows of V**H are
returned in the arrays U and VT;
 = 'S':  the first min(M,N) columns of U and the first
min(M,N) rows of V**H are returned in the arrays U
and VT;
 = 'O':  If M  > = N, the first N columns of U are overwritten
on the array A and all rows of V**H are returned in
the array VT;
otherwise, all columns of U are returned in the
array U and the first M rows of V**H are overwritten
in the array VT;
 = 'N':  no columns of U or rows of V**H are computed.
```

- **M (input)**
  The number of rows of the input matrix A. M >= 0.
- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBZ = 'O', A is overwritten with the first N columns of U (the left singular vectors, stored columnwise) if M >= N; A is overwritten with the first M rows of V**H (the right singular vectors, stored rowwise) otherwise. if JOBZ .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **S (output)**
  The singular values of A, sorted so that `S(i)` >= S(i+1).
- **U (output)**
  UCOL = M if JOBZ = 'A' or JOBZ = 'O' and M < N; UCOL = `min(M,N)` if JOBZ = 'S'. If JOBZ = 'A' or JOBZ = 'O' and M < N, U contains the M-by-M unitary matrix U; if JOBZ = 'S', U contains the first `min(M,N)` columns of U (the left singular vectors, stored columnwise); if JOBZ = 'O' and M >= N, or JOBZ = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBZ = 'S' or 'A' or JOBZ = 'O' and M < N, LDU >= M.
- **VT (output)**
  If JOBZ = 'A' or JOBZ = 'O' and M >= N, VT contains the N-by-N unitary matrix V**H; if JOBZ = 'S', VT contains the first `min(M,N)` rows of V**H (the right singular vectors, stored rowwise); if JOBZ = 'O' and M < N, or JOBZ = 'N', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBZ = 'A' or JOBZ = 'O' and M >= N, LDVT >= N; if JOBZ = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. if JOBZ = 'N', LWORK >= 2*min(M,N)+max(M,N). if JOBZ = 'O', LWORK >= 2*min(M,N)*min(M,N)+2*min(M,N)+max(M,N). if JOBZ = 'S' or 'A', LWORK >= min(M,N)*min(M,N)+2*min(M,N)+max(M,N). For good performance, LWORK should generally be larger. If LWORK < 0 but other input arguments are legal, `WORK(1)` returns optimal LWORK.
- **RWORK (workspace)**
  If JOBZ = 'N', LRWORK >= 7*min(M,N). Otherwise, LRWORK >= 5*min(M,N)*min(M,N) + 5*min(M,N)
- **IWORK (workspace)**
  `dimension(8*MIN(M,N))`
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.

> 0:  The updating process of SBDSDC did not converge.
```

FURTHER DETAILS

Based on contributions by

Ming Gu and Huan Ren, Computer Science Division, University of California at Berkeley, USA

NAME

zgesv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZGESV( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGESV_64( N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GESV( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GESV_64( [N], [NRHS], A, [LDA], IPIVOT, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgesv(int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zgesv_64(long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zgesv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as

```
A = P * L * U,
```

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the N-by-N coefficient matrix A. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the N-by-NRHS matrix of right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero.  The factorization
has been completed, but the factor U is exactly
singular, so the solution could not be computed.
```

NAME

zgesvd - compute the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors

---

SYNOPSIS

```
 SUBROUTINE ZGESVD( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBU, JOBVT
 DOUBLE COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER M, N, LDA, LDU, LDVT, LDWORK, INFO
 DOUBLE PRECISION SING(*), WORK2(*)

 SUBROUTINE ZGESVD_64( JOBU, JOBVT, M, N, A, LDA, SING, U, LDU, VT,
*      LDVT, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBU, JOBVT
 DOUBLE COMPLEX A(LDA,*), U(LDU,*), VT(LDVT,*), WORK(*)
 INTEGER*8 M, N, LDA, LDU, LDVT, LDWORK, INFO
 DOUBLE PRECISION SING(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVD( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, U, VT
 INTEGER :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL(8), DIMENSION(:) :: SING, WORK2

 SUBROUTINE GESVD_64( JOBU, JOBVT, [M], [N], A, [LDA], SING, U, [LDU],
*       VT, [LDVT], [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBVT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, U, VT
 INTEGER(8) :: M, N, LDA, LDU, LDVT, LDWORK, INFO
 REAL(8), DIMENSION(:) :: SING, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgesvd(char jobu, char jobvt, int m, int n, doublecomplex *a, int lda, double *sing, doublecomplex *u, int ldu, doublecomplex *vt, int ldvt, int *info);

void zgesvd_64(char jobu, char jobvt, long m, long n, doublecomplex *a, long lda, double *sing, doublecomplex *u, long ldu, doublecomplex *vt, long ldvt, long *info);

PURPOSE

zgesvd computes the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors. The SVD is written = U * SIGMA * conjugate-transpose(V)

where SIGMA is an M-by-N matrix which is zero except for its `min(m,n)` diagonal elements, U is an M-by-M unitary matrix, and V is an N-by-N unitary matrix. The diagonal elements of SIGMA are the singular values of A; they are real and non-negative, and are returned in descending order. The first `min(m,n)` columns of U and V are the left and right singular vectors of A.

Note that the routine returns V**H, not V.

ARGUMENTS

- **JOBU (input)**
  Specifies options for computing all or part of the matrix U:

  ```
   = 'A':  all M columns of U are returned in array U:

   = 'S':  the first min(m,n) columns of U (the left singular
  vectors) are returned in the array U;
   = 'O':  the first min(m,n) columns of U (the left singular
  vectors) are overwritten on the array A;
   = 'N':  no columns of U (no left singular vectors) are
  computed.
  ```

- **JOBVT (input)**
  Specifies options for computing all or part of the matrix V**H:

  ```
   = 'A':  all N rows of V**H are returned in the array VT;

   = 'S':  the first min(m,n) rows of V**H (the right singular
  vectors) are returned in the array VT;
   = 'O':  the first min(m,n) rows of V**H (the right singular
  vectors) are overwritten on the array A;
   = 'N':  no rows of V**H (no right singular vectors) are
  computed.
  ```

  JOBVT and JOBU cannot both be 'O'.

- **M (input)**

The number of rows of the input matrix A. M >= 0.

- **N (input)**
  The number of columns of the input matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if JOBU = 'O', A is overwritten with the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBVT = 'O', A is overwritten with the first `min(m,n)` rows of V**H (the right singular vectors, stored rowwise); if JOBU .ne. 'O' and JOBVT .ne. 'O', the contents of A are destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **SING (output)**
  The singular values of A, sorted so that `SING(i)` >= SING(i+1).
- **U (output)**
  (LDU,M) if JOBU = 'A' or (LDU,min(M,N)) if JOBU = 'S'. If JOBU = 'A', U contains the M-by-M unitary matrix U; if JOBU = 'S', U contains the first `min(m,n)` columns of U (the left singular vectors, stored columnwise); if JOBU = 'N' or 'O', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= 1; if JOBU = 'S' or 'A', LDU >= M.
- **VT (output)**
  If JOBVT = 'A', VT contains the N-by-N unitary matrix V**H; if JOBVT = 'S', VT contains the first `min(m,n)` rows of V**H (the right singular vectors, stored rowwise); if JOBVT = 'N' or 'O', VT is not referenced.
- **LDVT (input)**
  The leading dimension of the array VT. LDVT >= 1; if JOBVT = 'A', LDVT >= N; if JOBVT = 'S', LDVT >= min(M,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= 1. LDWORK >= 2*MIN(M,N)+MAX(M,N) For good performance, LDWORK should generally be larger.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  DIMENSION(5*MIN(M,N)). On exit, if INFO > 0, `WORK2(1:MIN(M,N)-1)` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in SING (not necessarily sorted). B satisfies A = U * B * VT, so it has the same singular values as A, and singular vectors related by U and VT.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if CBDSQR did not converge, INFO specifies how many
superdiagonals of an intermediate bidiagonal form B
did not converge to zero. See the description of WORK2
above for details.
```

NAME

zgesvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZGESVX( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR, WORK,
*      WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGESVX_64( FACT, TRANSA, N, NRHS, A, LDA, AF, LDAF,
*      IPIVOT, EQUED, ROWSC, COLSC, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION ROWSC(*), COLSC(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GESVX( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

```
 SUBROUTINE GESVX_64( FACT, [TRANSA], [N], [NRHS], A, [LDA], AF,
*       [LDAF], IPIVOT, EQUED, ROWSC, COLSC, B, [LDB], X, [LDX], RCOND,
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: ROWSC, COLSC, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgesvx(char fact, char transa, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, char equed, double *rowsc, double *colsc, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zgesvx_64(char fact, char transa, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, char equed, double *rowsc, double *colsc, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zgesvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
    TRANS = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
    TRANS = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
    TRANS = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if TRANS='N')
or diag(C)*B (if TRANS = 'T' or 'C').
```

2. If FACT = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as

```
    A = P * L * U,

where P is a permutation matrix, L is a unit lower triangular
matrix, and U is upper triangular.
```

3. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(C)` (if TRANS = 'N') or `diag(R)` (if TRANS = 'T' or 'C') so that it solves the original system before equilibration.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF and IPIVOT contain the factored form of A. If EQUED is not 'N', the matrix A has been equilibrated with scaling factors given by ROWSC and COLSC. A, AF, and IPIVOT are not modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
   = 'N':  A * X  = B     (No transpose)

   = 'T':  A**T * X  = B  (Transpose)

   = 'COLSC':  A**H * X  = B  (Conjugate transpose)
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the N-by-N matrix A. If FACT = 'F' and EQUED is not 'N', then A must have been equilibrated by the scaling factors in ROWSC and/or COLSC. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if EQUED .ne. 'N', A is scaled as follows: EQUED = 'ROWSC': A : = `diag(ROWSC)` * A

  EQUED = 'COLSC': A : = A * `diag(COLSC)`

  EQUED = 'B': A : = `diag(ROWSC)` * A * diag(COLSC).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the factors L and U from the factorization A = P*L*U as computed by CGETRF. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the factors L and U from the factorization A = P*L*U of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**

The leading dimension of the array AF. LDAF > = max(1,N).

- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the factorization A = P*L*U as computed by CGETRF; row i of the matrix was interchanged with row IPIVOT(i).

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the original matrix A.

  If FACT = 'E', then IPIVOT is an output argument and on exit contains the pivot indices from the factorization A = P*L*U of the equilibrated matrix A.

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'ROWSC':  Row equilibration, i.e., A has been premultiplied by
diag(ROWSC).
 = 'COLSC':  Column equilibration, i.e., A has been postmultiplied
by diag(COLSC).
 = 'B':  Both row and column equilibration, i.e., A has been
replaced by diag(ROWSC) * A * diag(COLSC).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **ROWSC (input/output)**
  The row scale factors for A. If EQUED = 'ROWSC' or 'B', A is multiplied on the left by diag(ROWSC); if EQUED = 'N' or 'COLSC', ROWSC is not accessed. ROWSC is an input argument if FACT = 'F'; otherwise, ROWSC is an output argument. If FACT = 'F' and EQUED = 'ROWSC' or 'B', each element of ROWSC must be positive.

- **COLSC (input/output)**
  The column scale factors for A. If EQUED = 'COLSC' or 'B', A is multiplied on the right by diag(COLSC); if EQUED = 'N' or 'ROWSC', COLSC is not accessed. COLSC is an input argument if FACT = 'F'; otherwise, COLSC is an output argument. If FACT = 'F' and EQUED = 'COLSC' or 'B', each element of COLSC must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if TRANSA = 'N' and EQUED = 'ROWSC' or 'B', B is overwritten by diag(ROWSC)*B; if TRANSA = 'T' or 'COLSC' and EQUED = 'COLSC' or 'B', B is overwritten by diag(COLSC)*B.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that A and B are modified on exit if EQUED .ne. 'N', and the solution to the equilibrated system is inv(diag(COLSC))*X if TRANSA = 'N' and EQUED = 'COLSC' or 'B', or inv(diag(ROWSC))*X if TRANSA = 'T' or 'COLSC' and EQUED = 'ROWSC' or 'B'.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).

- **WORK (workspace)**

dimension(2*N)

- **WORK2 (workspace)**
  dimension(2*N) On exit, WORK2(1) contains the reciprocal pivot growth factor norm(A)/norm(U). The ``max absolute element'' norm is used. If WORK2(1) is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X, condition estimator RCOND, and forward error bound FERR could be unreliable. If factorization fails with 0 <INFO < =N, then WORK2(1) contains the reciprocal pivot growth factor for the leading INFO columns of A.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization has
been completed, but the factor U is exactly
singular, so the solution and error bounds
could not be computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zgetf2 - compute an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE ZGETF2( M, N, A, LDA, IPIV, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER M, N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE ZGETF2_64( M, N, A, LDA, IPIV, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE GETF2( [M], [N], A, [LDA], IPIV, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE GETF2_64( [M], [N], A, [LDA], IPIV, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zgetf2(int m, int n, doublecomplex *a, int lda, int *ipiv, int *info);

void zgetf2_64(long m, long n, doublecomplex *a, long lda, long *ipiv, long *info);

PURPOSE

zgetf2 computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 2 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the m by n matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **IPIV (output)**
  The pivot indices; for 1 < = i < = min(M,N), row i of the matrix was interchanged with row IPIV(i).
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, U(k,k) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

zgetrf - compute an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges

---

SYNOPSIS

```
SUBROUTINE ZGETRF( M, N, A, LDA, IPIVOT, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER M, N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGETRF_64( M, N, A, LDA, IPIVOT, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 M, N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRF( [M], [N], A, [LDA], IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRF_64( [M], [N], A, [LDA], IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgetrf(int m, int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zgetrf_64(long m, long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zgetrf computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges.

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

This is the right-looking Level 3 BLAS version of the algorithm.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **IPIVOT (output)**
  The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIVOT(i).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero. The factorization
has been completed, but the factor U is exactly
singular, and division by zero will occur if it is used
to solve a system of equations.
```

NAME

zgetri - compute the inverse of a matrix using the LU factorization computed by CGETRF

SYNOPSIS

```
SUBROUTINE ZGETRI( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGETRI_64( N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRI( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRI_64( [N], A, [LDA], IPIVOT, [WORK], [LDWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgetri(int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zgetri_64(long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zgetri computes the inverse of a matrix using the LU factorization computed by CGETRF.

This method inverts U and then computes `inv(A)` by solving the system inv(A)*L = `inv(U)` for inv(A).

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the factors L and U from the factorization A = P*L*U as computed by CGETRF. On exit, if INFO = 0, the inverse of the original matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 < =i < =N, row i of the matrix was interchanged with row IPIVOT(i).
- **WORK (workspace)**
  On exit, if INFO =0, then `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,N). For optimal performance LDWORK > = N*NB, where NB is the optimal blocksize returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero; the matrix is
singular and its inverse could not be computed.
```

NAME

zgetrs - solve a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF

SYNOPSIS

```
SUBROUTINE ZGETRS( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGETRS_64( TRANSA, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 TRANSA
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GETRS( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: TRANSA
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GETRS_64( [TRANSA], [N], [NRHS], A, [LDA], IPIVOT, B,
*       [LDB], [INFO])
CHARACTER(LEN=1) :: TRANSA
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgetrs(char transa, int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zgetrs_64(char transa, long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zgetrs solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input)**
  The factors L and U from the factorization A = P*L*U as computed by CGETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (input)**
  The pivot indices from CGETRF; for 1 <=i <=N, row i of the matrix was interchanged with row IPIVOT(i).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zggbak - form the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL

SYNOPSIS

```
 SUBROUTINE ZGGBAK( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 DOUBLE COMPLEX V(LDV,*)
 INTEGER N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*)

 SUBROUTINE ZGGBAK_64( JOB, SIDE, N, ILO, IHI, LSCALE, RSCALE, M, V,
*      LDV, INFO)
 CHARACTER * 1 JOB, SIDE
 DOUBLE COMPLEX V(LDV,*)
 INTEGER*8 N, ILO, IHI, M, LDV, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAK( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M], V,
*       [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX(8), DIMENSION(:,:) :: V
 INTEGER :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE

 SUBROUTINE GGBAK_64( JOB, SIDE, [N], ILO, IHI, LSCALE, RSCALE, [M],
*       V, [LDV], [INFO])
 CHARACTER(LEN=1) :: JOB, SIDE
 COMPLEX(8), DIMENSION(:,:) :: V
 INTEGER(8) :: N, ILO, IHI, M, LDV, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE
```

C INTERFACE

#include <sunperf.h>

void zggbak(char job, char side, int n, int ilo, int ihi, double *lscale, double *rscale, int m, doublecomplex *v, int ldv, int *info);

void zggbak_64(char job, char side, long n, long ilo, long ihi, double *lscale, double *rscale, long m, doublecomplex *v, long ldv, long *info);

PURPOSE

zggbak forms the right or left eigenvectors of a complex generalized eigenvalue problem A*x = lambda*B*x, by backward transformation on the computed eigenvectors of the balanced pair of matrices output by CGGBAL.

ARGUMENTS

- **JOB (input)**
  Specifies the type of backward transformation required:

  ```
  = 'N':  do nothing, return immediately;

  = 'P':  do backward transformation for permutation only;

  = 'S':  do backward transformation for scaling only;

  = 'B':  do backward transformations for both permutation and
scaling.
JOB must be the same as the argument JOB supplied to CGGBAL.
```

- **SIDE (input)**

  ```
  = 'R':  V contains right eigenvectors;

  = 'L':  V contains left eigenvectors.
```

- **N (input)**
  The number of rows of the matrix V. N > = 0.
- **ILO (input)**
  The integers ILO and IHI determined by CGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  The integers ILO and IHI determined by CGGBAL. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **LSCALE (input)**
  Details of the permutations and/or scaling factors applied to the left side of A and B, as returned by CGGBAL.
- **RSCALE (input)**
  Details of the permutations and/or scaling factors applied to the right side of A and B, as returned by CGGBAL.
- **M (input)**
  The number of columns of the matrix V. M > = 0.

- **V (input/output)**
  On entry, the matrix of right or left eigenvectors to be transformed, as returned by CTGEVC. On exit, V is overwritten by the transformed eigenvectors.
- **LDV (input)**
  The leading dimension of the matrix V. LDV >= max(1,N).
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. Ward, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

zggbal - balance a pair of general complex matrices (A,B)

SYNOPSIS

```
 SUBROUTINE ZGGBAL( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE, RSCALE,
*      WORK, INFO)
 CHARACTER * 1 JOB
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, LDA, LDB, ILO, IHI, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*), WORK(*)

 SUBROUTINE ZGGBAL_64( JOB, N, A, LDA, B, LDB, ILO, IHI, LSCALE,
*      RSCALE, WORK, INFO)
 CHARACTER * 1 JOB
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, LDA, LDB, ILO, IHI, INFO
 DOUBLE PRECISION LSCALE(*), RSCALE(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGBAL( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, LDA, LDB, ILO, IHI, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE, WORK

 SUBROUTINE GGBAL_64( JOB, [N], A, [LDA], B, [LDB], ILO, IHI, LSCALE,
*       RSCALE, [WORK], [INFO])
 CHARACTER(LEN=1) :: JOB
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, LDA, LDB, ILO, IHI, INFO
 REAL(8), DIMENSION(:) :: LSCALE, RSCALE, WORK
```

C INTERFACE

#include <sunperf.h>

void zggbal(char job, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *ilo, int *ihi, double *lscale, double *rscale, int *info);

void zggbal_64(char job, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *ilo, long *ihi, double *lscale, double *rscale, long *info);

PURPOSE

zggbal balances a pair of general complex matrices (A,B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to ILO$-$1 and last IHI+1 to N elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ILO to IHI to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem A*x = lambda*B*x.

ARGUMENTS

- **JOB (input)**
  Specifies the operations to be performed on A and B:

```
 = 'N':  none:  simply set ILO  = 1, IHI  = N, LSCALE(I)  = 1.0
and RSCALE(I)  = 1.0 for i =1,...,N;
 = 'P':  permute only;

 = 'S':  scale only;

 = 'B':  both permute and scale.
```

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the input matrix A. On exit, A is overwritten by the balanced matrix. If JOB = 'N', A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the input matrix B. On exit, B is overwritten by the balanced matrix. If JOB = 'N', B is not referenced.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ILO (output)**
  ILO and IHI are set to integers such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If JOB = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  ILO and IHI are set to integers such that on exit A(i,j) = 0 and B(i,j) = 0 if i > j and j = 1,...,ILO-1 or i =

IHI+1,...,N.

- **LSCALE (input)**
  Details of the permutations and scaling factors applied to the left side of A and B. If `P(j)` is the index of the row interchanged with row j, and `D(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (input)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `P(j)` is the index of the column interchanged with column j, and `D(j)` is the scaling factor applied to column j, then `RSCALE(j)` = `P(j)` for J = 1,...,ILO-1 = `D(j)` for J = ILO,...,IHI = `P(j)` for J = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **WORK (workspace)**
  `dimension(6*N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

See R.C. WARD, Balancing the generalized eigenvalue problem, SIAM J. Sci. Stat. Comp. 2 (1981), 141-152.

NAME

zgges - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR)

SYNOPSIS

```
 SUBROUTINE ZGGES( JOBVSL, JOBVSR, SORT, DELZTG, N, A, LDA, B, LDB,
*      SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK, RWORK,
*      BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL DELZTG
 LOGICAL BWORK(*)
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZGGES_64( JOBVSL, JOBVSR, SORT, DELZTG, N, A, LDA, B,
*      LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, WORK, LWORK,
*      RWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL*8 DELZTG
 LOGICAL*8 BWORK(*)
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGES( JOBVSL, JOBVSR, SORT, DELZTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK],
*       [LWORK], [RWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
 LOGICAL :: DELZTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE GGES_64( JOBVSL, JOBVSR, SORT, DELZTG, [N], A, [LDA], B,
*       [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], [WORK],
*       [LWORK], [RWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT
```

```
COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, INFO
LOGICAL(8) :: DELZTG
LOGICAL(8), DIMENSION(:) :: BWORK
REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zgges(char jobvsl, char jobvsr, char sort, logical(*delztg)(COMPLEX*16,COMPLEX*16), int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *sdim, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, int ldvsl, doublecomplex *vsr, int ldvsr, int *info);

void zgges_64(char jobvsl, char jobvsr, char sort, logical(*delztg)(COMPLEX*16,COMPLEX*16), long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *sdim, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, long ldvsl, doublecomplex *vsr, long ldvsr, long *info);

PURPOSE

zgges computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized complex Schur form (S, T), and optionally left and/or right Schur vectors (VSL and VSR). This gives the generalized Schur factorization

```
(A,B) = ( (VSL)*S*(VSR)**H, (VSL)*T*(VSR)**H )
```

where (VSR)**H is the conjugate-transpose of VSR.

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper triangular matrix S and the upper triangular matrix T. The leading columns of VSL and VSR then form an unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver CGGEV instead, which is faster.)

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal elements of T are non-negative real numbers.

ARGUMENTS

- **JOBVSL (input)**

  ```
  = 'N':  do not compute the left Schur vectors;

  = 'V':  compute the left Schur vectors.
  ```

- **JOBVSR (input)**

  ```
  = 'N':  do not compute the right Schur vectors;

  = 'V':  compute the right Schur vectors.
  ```

- **SORT (input)**

Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S':  Eigenvalues are ordered (see DELZTG).
```

- **DELZTG (input)**
  DELZTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', DELZTG is not referenced. If SORT = 'S', DELZTG is used to select eigenvalues to sort to the top left of the Schur form. An eigenvalue `ALPHA(j)/BETA(j)` is selected if `DELZTG(ALPHA(j),BETA(j))` is true.

  Note that a selected complex eigenvalue may no longer satisfy `DELZTG(ALPHA(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+2 (See INFO below).
- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which DELZTG is true.
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. ALPHA(j), j =1,...,N and BETA(j), j =1,...,N are the diagonals of the complex Schur form (A,B) output by CGGES. The `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See description of ALPHA.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > = 1, and if JOBVSL = 'V', LDVSL > = N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR > = 1, and if JOBVSR = 'V', LDVSR > = N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,2*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(8*N)`
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
= 0:  successful exit
```

 < 0: if INFO = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed. (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
 > N: =N+1: other than QZ iteration failed in CHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy DELZTG =.TRUE. This could also
be caused due to scaling.
 =N+3: reordering falied in CTGSEN.

NAME

zggesx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T),

---

SYNOPSIS

```
 SUBROUTINE ZGGESX( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, N, A, LDA,
*      B, LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, RCONDE,
*      RCONDV, WORK, LWORK, RWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL DELCTG
 LOGICAL BWORK(*)
 DOUBLE PRECISION RCONDE(*), RCONDV(*), RWORK(*)

 SUBROUTINE ZGGESX_64( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, N, A,
*      LDA, B, LDB, SDIM, ALPHA, BETA, VSL, LDVSL, VSR, LDVSR, RCONDE,
*      RCONDV, WORK, LWORK, RWORK, IWORK, LIWORK, BWORK, INFO)
 CHARACTER * 1 JOBVSL, JOBVSR, SORT, SENSE
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VSL(LDVSL,*), VSR(LDVSR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 DELCTG
 LOGICAL*8 BWORK(*)
 DOUBLE PRECISION RCONDE(*), RCONDV(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGESX( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, [N], A, [LDA],
*       B, [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR], RCONDE,
*       RCONDV, [WORK], [LWORK], [RWORK], [IWORK], [LIWORK], [BWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL :: DELCTG
 LOGICAL, DIMENSION(:) :: BWORK
 REAL(8), DIMENSION(:) :: RCONDE, RCONDV, RWORK
```

```
 SUBROUTINE GGESX_64( JOBVSL, JOBVSR, SORT, DELCTG, SENSE, [N], A,
*       [LDA], B, [LDB], SDIM, ALPHA, BETA, VSL, [LDVSL], VSR, [LDVSR],
*       RCONDE, RCONDV, [WORK], [LWORK], [RWORK], [IWORK], [LIWORK],
*       [BWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVSL, JOBVSR, SORT, SENSE
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VSL, VSR
 INTEGER(8) :: N, LDA, LDB, SDIM, LDVSL, LDVSR, LWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8) :: DELCTG
 LOGICAL(8), DIMENSION(:) :: BWORK
 REAL(8), DIMENSION(:) :: RCONDE, RCONDV, RWORK
```

C INTERFACE

#include <sunperf.h>

void zggesx(char jobvsl, char jobvsr, char sort, logical(*delctg)(COMPLEX*16,COMPLEX*16), char sense, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *sdim, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, int ldvsl, doublecomplex *vsr, int ldvsr, double *rconde, double *rcondv, int *info);

void zggesx_64(char jobvsl, char jobvsr, char sort, logical(*delctg)(COMPLEX*16,COMPLEX*16), char sense, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *sdim, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vsl, long ldvsl, doublecomplex *vsr, long ldvsr, double *rconde, double *rcondv, long *info);

PURPOSE

zggesx computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, the complex Schur form (S,T), and, optionally, the left and/or right matrices of Schur vectors (VSL and VSR). This gives the generalized Schur factorization A,B) = ((VSL) S (VSR)**H, (VSL) T (VSR)**H)

where (VSR)**H is the conjugate-transpose of VSR.

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper triangular matrix S and the upper triangular matrix T; computes a reciprocal condition number for the average of the selected eigenvalues (RCONDE); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (RCONDV). The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio alpha/beta = w, such that A - w*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0 or for both being zero.

A pair of matrices (S,T) is in generalized complex Schur form if T is upper triangular with non-negative diagonal and S is upper triangular.

ARGUMENTS

- **JOBVSL (input)**

  = 'N': do not compute the left Schur vectors;

  = 'V': compute the left Schur vectors.
- **JOBVSR (input)**

  = 'N': do not compute the right Schur vectors;

```
= 'V':  compute the right Schur vectors.
```

- **SORT (input)**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered;

```
= 'S':  Eigenvalues are ordered (see DELCTG).
```

- **DELCTG (input)**
  DELCTG must be declared EXTERNAL in the calling subroutine. If SORT = 'N', DELCTG is not referenced. If SORT = 'S', DELCTG is used to select eigenvalues to sort to the top left of the Schur form. Note that a selected complex eigenvalue may no longer satisfy `DELCTG(ALPHA(j),BETA(j))` = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case INFO is set to N+3 see INFO below).
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N' : None are computed;

```
= 'E' : Computed for average of selected eigenvalues only;

= 'V' : Computed for selected deflating subspaces only;

= 'B' : Computed for both.
If SENSE  = 'E', 'V', or 'B', SORT must equal 'S'.
```

- **N (input)**
  The order of the matrices A, B, VSL, and VSR. N > = 0.
- **A (input/output)**
  On entry, the first of the pair of matrices. On exit, A has been overwritten by its generalized Schur form S.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the second of the pair of matrices. On exit, B has been overwritten by its generalized Schur form T.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **SDIM (output)**
  If SORT = 'N', SDIM = 0. If SORT = 'S', SDIM = number of eigenvalues (after sorting) for which DELCTG is true.
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues. `ALPHA(j)` and BETA(j),j =1,...,N are the diagonals of the complex Schur form (S,T). `BETA(j)` will be non-negative real.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See description of ALPHA.
- **VSL (output)**
  If JOBVSL = 'V', VSL will contain the left Schur vectors. Not referenced if JOBVSL = 'N'.
- **LDVSL (input)**
  The leading dimension of the matrix VSL. LDVSL > =1, and if JOBVSL = 'V', LDVSL > = N.
- **VSR (output)**
  If JOBVSR = 'V', VSR will contain the right Schur vectors. Not referenced if JOBVSR = 'N'.
- **LDVSR (input)**
  The leading dimension of the matrix VSR. LDVSR > = 1, and if JOBVSR = 'V', LDVSR > = N.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', `RCONDE(1)` and `RCONDE(2)` contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if SENSE = 'N' or 'V'.
- **RCONDV (output)**
  If SENSE = 'V' or 'B', `RCONDV(1)` and `RCONDV(2)` contain the reciprocal condition number for the selected deflating subspaces. Not referenced if SENSE = 'N' or 'E'.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 2*N. If SENSE = 'E', 'V', or 'B', LWORK > = MAX(2*N, 2*SDIM*(N-SDIM)).
- **RWORK (workspace)**
  `dimension(8*N)` Real workspace.
- **IWORK (workspace)**
  Not referenced if SENSE = 'N'. On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array WORK. LIWORK > = N+2.
- **BWORK (workspace)**
  `dimension(N)` Not referenced if SORT = 'N'.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  (A,B) are not in Schur
form, but ALPHA(j) and BETA(j) should be correct for
j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in CHGEQZ

 =N+2: after reordering, roundoff changed values of
some complex eigenvalues so that leading
eigenvalues in the Generalized Schur form no
longer satisfy DELCTG =.TRUE.  This could also
be caused due to scaling.
 =N+3: reordering failed in CTGSEN.
```

NAME

zggev - compute for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

---

SYNOPSIS

```
 SUBROUTINE ZGGEV( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA, VL,
*      LDVL, VR, LDVR, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZGGEV_64( JOBVL, JOBVR, N, A, LDA, B, LDB, ALPHA, BETA,
*      VL, LDVL, VR, LDVR, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOBVL, JOBVR
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGEV( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA, BETA,
*       VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE GGEV_64( JOBVL, JOBVR, [N], A, [LDA], B, [LDB], ALPHA,
*       BETA, VL, [LDVL], VR, [LDVR], [WORK], [LWORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBVL, JOBVR
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, LWORK, INFO
 REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zggev(char jobvl, char jobvr, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int *info);

void zggev_64(char jobvl, char jobvr, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long *info);

PURPOSE

zggev computes for a pair of N-by-N complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right generalized eigenvector `v(j)` corresponding to the generalized eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j).
```

The left generalized eigenvector `u(j)` corresponding to the generalized eigenvalues `lambda(j)` of (A,B) satisfies

```
u(j)**H * A = lambda(j) * u(j)**H * B
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```

- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```

- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.

- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten.

- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten.

- **LDB (input)**
  The leading dimension of B. LDB >= max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues.

  Note: the quotients `ALPHA(j)/BETA(j)` may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio alpha/beta. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).
- **BETA (output)**
  See description of ALPHA.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,2*N). For good performance, LWORK must generally be larger.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(8*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 =1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and BETA(j) should be
correct for j =INFO+1,...,N.
 > N:   =N+1: other then QZ iteration failed in SHGEQZ,

 =N+2: error return from STGEVC.
```

NAME

zggevx - compute for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors

---

SYNOPSIS

```
 SUBROUTINE ZGGEVX( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B, LDB,
*      ALPHA, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE, RSCALE, ABNRM,
*      BBNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, IWORK, BWORK, INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL BWORK(*)
 DOUBLE PRECISION ABNRM, BBNRM
 DOUBLE PRECISION LSCALE(*), RSCALE(*), RCONDE(*), RCONDV(*), RWORK(*)

 SUBROUTINE ZGGEVX_64( BALANC, JOBVL, JOBVR, SENSE, N, A, LDA, B,
*      LDB, ALPHA, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSCALE, RSCALE,
*      ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, IWORK, BWORK,
*      INFO)
 CHARACTER * 1 BALANC, JOBVL, JOBVR, SENSE
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 BWORK(*)
 DOUBLE PRECISION ABNRM, BBNRM
 DOUBLE PRECISION LSCALE(*), RSCALE(*), RCONDE(*), RCONDV(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGEVX( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
*       [LDB], ALPHA, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI, LSCALE,
*       RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK], [RWORK],
*       [IWORK], [BWORK], [INFO])
 CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
```

```
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL, DIMENSION(:) :: BWORK
  REAL(8) :: ABNRM, BBNRM
  REAL(8), DIMENSION(:) :: LSCALE, RSCALE, RCONDE, RCONDV, RWORK

  SUBROUTINE GGEVX_64( BALANC, JOBVL, JOBVR, SENSE, [N], A, [LDA], B,
 *       [LDB], ALPHA, BETA, VL, [LDVL], VR, [LDVR], ILO, IHI, LSCALE,
 *       RSCALE, ABNRM, BBNRM, RCONDE, RCONDV, [WORK], [LWORK], [RWORK],
 *       [IWORK], [BWORK], [INFO])
  CHARACTER(LEN=1) :: BALANC, JOBVL, JOBVR, SENSE
  COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
  COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
  INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, ILO, IHI, LWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8), DIMENSION(:) :: BWORK
  REAL(8) :: ABNRM, BBNRM
  REAL(8), DIMENSION(:) :: LSCALE, RSCALE, RCONDE, RCONDV, RWORK
```

C INTERFACE

#include <sunperf.h>

void zggevx(char balanc, char jobvl, char jobvr, char sense, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int *ilo, int *ihi, double *lscale, double *rscale, double *abnrm, double *bbnrm, double *rconde, double *rcondv, int *info);

void zggevx_64(char balanc, char jobvl, char jobvr, char sense, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long *ilo, long *ihi, double *lscale, double *rscale, double *abnrm, double *bbnrm, double *rconde, double *rcondv, long *info);

PURPOSE

zggevx computes for a pair of N-by-N complex nonsymmetric matrices (A,B) the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally, it also computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ILO, IHI, LSCALE, RSCALE, ABNRM, and BBNRM), reciprocal condition numbers for the eigenvalues (RCONDE), and reciprocal condition numbers for the right eigenvectors (RCONDV).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar lambda or a ratio alpha/beta = lambda, such that A - lambda*B is singular. It is usually represented as the pair (alpha,beta), as there is a reasonable interpretation for beta=0, and even for both being zero.

The right eigenvector `v(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
A * v(j) = lambda(j) * B * v(j) .
```

The left eigenvector `u(j)` corresponding to the eigenvalue `lambda(j)` of (A,B) satisfies

```
u(j)**H * A  = lambda(j) * u(j)**H * B.
```

where u(j)**H is the conjugate-transpose of u(j).

ARGUMENTS

- **BALANC (input)**
  Specifies the balance option to be performed:

  ```
  = 'N':  do not diagonally scale or permute;

  = 'P':  permute only;

  = 'S':  scale only;

  = 'B':  both permute and scale.
  ```
  Computed reciprocal condition numbers will be for the matrices after permuting and/or balancing. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
- **JOBVL (input)**

  ```
  = 'N':  do not compute the left generalized eigenvectors;

  = 'V':  compute the left generalized eigenvectors.
  ```
- **JOBVR (input)**

  ```
  = 'N':  do not compute the right generalized eigenvectors;

  = 'V':  compute the right generalized eigenvectors.
  ```
- **SENSE (input)**
  Determines which reciprocal condition numbers are computed. = 'N': none are computed;

  ```
  = 'E': computed for eigenvalues only;

  = 'V': computed for eigenvectors only;

  = 'B': computed for eigenvalues and eigenvectors.
  ```
- **N (input)**
  The order of the matrices A, B, VL, and VR. N > = 0.
- **A (input/output)**
  On entry, the matrix A in the pair (A,B). On exit, A has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then A contains the first part of the complex Schur form of the ``balanced'' versions of the input A and B.
- **LDA (input)**
  The leading dimension of A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the matrix B in the pair (A,B). On exit, B has been overwritten. If JOBVL ='V' or JOBVR ='V' or both, then B contains the second part of the complex Schur form of the ``balanced'' versions of the input A and B.
- **LDB (input)**
  The leading dimension of B. LDB > = max(1,N).
- **ALPHA (output)**
  On exit, ALPHA(j)/BETA(j), j =1,...,N, will be the generalized eigenvalues.

  Note: the quotient `ALPHA(j)/BETA(j)`) may easily over- or underflow, and `BETA(j)` may even be zero. Thus, the user should avoid naively computing the ratio ALPHA/BETA. However, ALPHA will be always less than and usually comparable with `norm(A)` in magnitude, and BETA always less than and usually comparable with norm(B).

- **BETA (output)**
  See description of ALPHA.
- **VL (output)**
  If JOBVL = 'V', the left generalized eigenvectors `u(j)` are stored one after another in the columns of VL, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVL = 'N'.
- **LDVL (input)**
  The leading dimension of the matrix VL. LDVL >= 1, and if JOBVL = 'V', LDVL >= N.
- **VR (output)**
  If JOBVR = 'V', the right generalized eigenvectors `v(j)` are stored one after another in the columns of VR, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component will have abs(real part) + abs(imag. part) = 1. Not referenced if JOBVR = 'N'.
- **LDVR (input)**
  The leading dimension of the matrix VR. LDVR >= 1, and if JOBVR = 'V', LDVR >= N.
- **ILO (output)**
  ILO is an integer value such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **IHI (output)**
  IHI is an integer value such that on exit `A(i,j)` = 0 and `B(i,j)` = 0 if i > j and j = 1,...,ILO-1 or i = IHI+1,...,N. If BALANC = 'N' or 'S', ILO = 1 and IHI = N.
- **LSCALE (output)**
  Details of the permutations and scaling factors applied to the left side of A and B. If `PL(j)` is the index of the row interchanged with row j, and `DL(j)` is the scaling factor applied to row j, then `LSCALE(j)` = `PL(j)` for j = 1,...,ILO-1 = `DL(j)` for j = ILO,...,IHI = `PL(j)` for j = IHI+1,...,N. The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **RSCALE (output)**
  Details of the permutations and scaling factors applied to the right side of A and B. If `PR(j)` is the index of the column interchanged with column j, and `DR(j)` is the scaling factor applied to column j, then `RSCALE(j)` = `PR(j)` for j = 1,...,ILO-1 = `DR(j)` for j = ILO,...,IHI = `PR(j)` for j = IHI+1,...,N The order in which the interchanges are made is N to IHI+1, then 1 to ILO-1.
- **ABNRM (output)**
  The one-norm of the balanced matrix A.
- **BBNRM (output)**
  The one-norm of the balanced matrix B.
- **RCONDE (output)**
  If SENSE = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If SENSE = 'V', RCONDE is not referenced.
- **RCONDV (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute RCONDV(j), `RCONDV(j)` is set to 0; this can only occur when the true value would be very small anyway. If SENSE = 'E', RCONDV is not referenced. Not referenced if JOB = 'E'.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,2*N). If SENSE = 'N' or 'E', LWORK >= 2*N. If SENSE = 'V' or 'B', LWORK >= 2*N*N+2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(6*N)` Real workspace.
- **IWORK (workspace)**
  `dimension(N+2)` If SENSE = 'E', IWORK is not referenced.
- **BWORK (workspace)**

dimension(N) If SENSE = 'N', BWORK is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 = 1,...,N:
The QZ iteration failed.  No eigenvectors have been
calculated, but ALPHA(j) and BETA(j) should be correct
for j =INFO+1,...,N.
 > N:   =N+1: other than QZ iteration failed in CHGEQZ.

 =N+2: error return from CTGEVC.
```

FURTHER DETAILS

Balancing a matrix pair (A,B) includes, first, permuting rows and columns to isolate eigenvalues, second, applying diagonal similarity transformation to the rows and columns to make the rows and columns as close in norm as possible. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.11.1.2 of LAPACK Users' Guide.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

hord(w, lambda) < = EPS * norm(ABNRM, BBNRM) / RCONDE(I)

An approximate error bound for the angle between the i-th computed eigenvector VL(i) or VR(i) is given by

PS * norm(ABNRM, BBNRM) / DIF(i).

For further explanation of the reciprocal condition numbers RCONDE and RCONDV, see section 4.11 of LAPACK User's Guide.

NAME

zggglm - solve a general Gauss-Markov linear model (GLM) problem

---

SYNOPSIS

```
 SUBROUTINE ZGGGLM( N, M, P, A, LDA, B, LDB, D, X, Y, WORK, LDWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
 INTEGER N, M, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE ZGGGLM_64( N, M, P, A, LDA, B, LDB, D, X, Y, WORK,
*      LDWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), D(*), X(*), Y(*), WORK(*)
 INTEGER*8 N, M, P, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGGLM( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y, [WORK],
*       [LDWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: D, X, Y, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, M, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE GGGLM_64( [N], [M], [P], A, [LDA], B, [LDB], D, X, Y,
*       [WORK], [LDWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: D, X, Y, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, M, P, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zggglm(int n, int m, int p, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *d, doublecomplex *x, doublecomplex *y, int *info);

void zggglm_64(long n, long m, long p, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *d, doublecomplex *x, doublecomplex *y, long *info);

PURPOSE

zggglm solves a general Gauss-Markov linear model (GLM) problem:

```
        minimize || y ||_2   subject to   d = A*x + B*y

            x
```

where A is an N-by-M matrix, B is an N-by-P matrix, and d is a given N-vector. It is assumed that M <= N <= M+P, and

```
           rank(A) = M    and    rank( A B ) = N.
```

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of A and B.

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

```
             minimize || inv(B)*(d-A*x) ||_2

                 x
```

where `inv(B)` denotes the inverse of B.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. 0 < = M < = N.
- **P (input)**
  The number of columns of the matrix B. P > = N-M.
- **A (input/output)**
  On entry, the N-by-M matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-P matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **D (input/output)**
  On entry, D is the left hand side of the GLM equation. On exit, D is destroyed.
- **X (output)**
  On exit, X and Y are the solutions of the GLM problem.

- **Y (output)**
  On exit, X and Y are the solutions of the GLM problem.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK >= max(1,N+M+P). For optimum performance, LDWORK >= M+min(N,P)+max(N,P)*NB, where NB is an upper bound for the optimal blocksizes for CGEQRF, CGERQF, CUNMQR and CUNMRQ.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

zgghrd - reduce a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular

---

SYNOPSIS

```
 SUBROUTINE ZGGHRD( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO

 SUBROUTINE ZGGHRD_64( COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB, Q,
*       LDQ, Z, LDZ, INFO)
 CHARACTER * 1 COMPQ, COMPZ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGHRD( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO

 SUBROUTINE GGHRD_64( COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B, [LDB],
*        Q, [LDQ], Z, [LDZ], [INFO])
 CHARACTER(LEN=1) :: COMPQ, COMPZ
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, INFO
```

C INTERFACE

#include <sunperf.h>

void zgghrd(char compq, char compz, int n, int ilo, int ihi, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *q, int ldq, doublecomplex *z, int ldz, int *info);

void zgghrd_64(char compq, char compz, long n, long ilo, long ihi, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *q, long ldq, doublecomplex *z, long ldz, long *info);

PURPOSE

zgghrd reduces a pair of complex matrices (A,B) to generalized upper Hessenberg form using unitary transformations, where A is a general matrix and B is upper triangular: Q' * A * Z = H and Q' * B * Z = T, where H is upper Hessenberg, T is upper triangular, and Q and Z are unitary, and ' means conjugate transpose.

The unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q1 and Z1, so that

1 * A * Z1' = (Q1*Q) * H * (Z1*Z)' 1 * B * Z1' = (Q1*Q) * T * (Z1*Z)'

ARGUMENTS

- **COMPQ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'V': Q must contain a unitary matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **COMPZ (input)**

```
 = 'N': do not compute Q;

 = 'I': Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'V': Q must contain a unitary matrix Q1 on entry,
and the product Q1*Q is returned.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGGBAL; otherwise they should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See description of ILO.
- **A (input/output)**

On entry, the N-by-N general matrix to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the rest is set to zero.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. On exit, the upper triangular matrix T = Q' B Z. The elements below the diagonal are set to zero.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **Q (input/output)**
  If COMPQ ='N': Q is not referenced.

  If COMPQ ='I': on entry, Q need not be set, and on exit it contains the unitary matrix Q, where Q' is the product of the Givens transformations which are applied to A and B on the left. If COMPQ ='V': on entry, Q must contain a unitary matrix Q1, and on exit this is overwritten by Q1*Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= N if COMPQ ='V' or 'I'; LDQ >= 1 otherwise.
- **Z (input/output)**
  If COMPZ ='N': Z is not referenced.

  If COMPZ ='I': on entry, Z need not be set, and on exit it contains the unitary matrix Z, which is the product of the Givens transformations which are applied to A and B on the right. If COMPZ ='V': on entry, Z must contain a unitary matrix Z1, and on exit this is overwritten by Z1*Z.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= N if COMPZ ='V' or 'I'; LDZ >= 1 otherwise.
- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

This routine reduces A to Hessenberg and B to triangular form by an unblocked reduction, as described in _Matrix_Computations_, by Golub and van Loan (Johns Hopkins Press).

NAME

zgglse - solve the linear equality-constrained least squares (LSE) problem

---

SYNOPSIS

```
 SUBROUTINE ZGGLSE( M, N, P, A, LDA, B, LDB, C, D, X, WORK, LDWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
 INTEGER M, N, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE ZGGLSE_64( M, N, P, A, LDA, B, LDB, C, D, X, WORK,
*      LDWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(*), D(*), X(*), WORK(*)
 INTEGER*8 M, N, P, LDA, LDB, LDWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGLSE( [M], [N], [P], A, [LDA], B, [LDB], C, D, X, [WORK],
*       [LDWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: C, D, X, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, P, LDA, LDB, LDWORK, INFO

 SUBROUTINE GGLSE_64( [M], [N], [P], A, [LDA], B, [LDB], C, D, X,
*       [WORK], [LDWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: C, D, X, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, P, LDA, LDB, LDWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zgglse(int m, int n, int p, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *c, doublecomplex *d, doublecomplex *x, int *info);

void zgglse_64(long m, long n, long p, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *c, doublecomplex *d, doublecomplex *x, long *info);

PURPOSE

zgglse solves the linear equality-constrained least squares (LSE) problem:

```
minimize || c - A*x ||_2   subject to   B*x = d
```

where A is an M-by-N matrix, B is a P-by-N matrix, c is a given M-vector, and d is a given P-vector. It is assumed that

P <= N <= M+P, and

```
rank(B) = P and  rank( ( A ) ) = N.
                       ( ( B ) )
```

These conditions ensure that the LSE problem has a unique solution, which is obtained using a GRQ factorization of the matrices B and A.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **P (input)**
  The number of rows of the matrix B. 0 < = P < = N < = M+P.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B is destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **C (input/output)**
  On entry, C contains the right hand side vector for the least squares part of the LSE problem. On exit, the residual sum of squares for the solution is given by the sum of squares of elements N-P+1 to M of vector C.
- **D (input/output)**
  On entry, D contains the right hand side vector for the constrained equation. On exit, D is destroyed.
- **X (output)**
  On exit, X is the solution of the LSE problem.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The dimension of the array WORK. LDWORK > = max(1,M+N+P). For optimum performance LDWORK > =

P+min(M,N)+max(M,N)*NB, where NB is an upper bound for the optimal blocksizes for CGEQRF, CGERQF, CUNMQR and CUNMRQ.

If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit.

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

zggqrf - compute a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B.

SYNOPSIS

```
 SUBROUTINE ZGGQRF( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER N, M, P, LDA, LDB, LWORK, INFO

 SUBROUTINE ZGGQRF_64( N, M, P, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER*8 N, M, P, LDA, LDB, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGQRF( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, M, P, LDA, LDB, LWORK, INFO

 SUBROUTINE GGQRF_64( [N], [M], [P], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, M, P, LDA, LDB, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zggqrf(int n, int m, int p, doublecomplex *a, int lda, doublecomplex *taua, doublecomplex *b, int ldb, doublecomplex

*taub, int *info);

void zggqrf_64(long n, long m, long p, doublecomplex *a, long lda, doublecomplex *taua, doublecomplex *b, long ldb, doublecomplex *taub, long *info);

PURPOSE

zggqrf computes a generalized QR factorization of an N-by-M matrix A and an N-by-P matrix B:

```
A = Q*R,        B = Q*T*Z,
```

where Q is an N-by-N unitary matrix, Z is a P-by-P unitary matrix, and R and T assume one of the forms:

if N >= M, R = (R11) M , or if N < M, R = (R11 R12) N, (0) N-M N M-N M

where R11 is upper triangular, and

if N <= P, T = (0 T12) N, or if N > P, T = (T11) N-P, P-N N (T21) P P

where T12 or T21 is upper triangular.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of inv(B)*A:

```
inv(B)*A = Z'*(inv(T)*R)
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the conjugate transpose of matrix Z.

ARGUMENTS

- **N (input)**
  The number of rows of the matrices A and B. N > = 0.
- **M (input)**
  The number of columns of the matrix A. M > = 0.
- **P (input)**
  The number of columns of the matrix B. P > = 0.
- **A (input)**
  On entry, the N-by-M matrix A. On exit, the elements on and above the diagonal of the array contain the min(N,M)-by-M upper trapezoidal matrix R (R is upper triangular if N > = M); the elements below the diagonal, with the array TAUA, represent the unitary matrix Q as a product of `min(N,M)` elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q (see Further Details).
- **B (input)**
  On entry, the N-by-P matrix B. On exit, if N < = P, the upper triangle of the subarray `B(1:N,P-N+1:P)` contains the N-by-N upper triangular matrix T; if N > P, the elements on and above the (N-P)-th subdiagonal contain the N-by-P upper trapezoidal matrix T; the remaining elements, with the array TAUB, represent the unitary matrix Z as

a product of elementary reflectors (see Further Details).

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the QR factorization of an N-by-M matrix, NB2 is the optimal blocksize for the RQ factorization of an N-by-P matrix, and NB3 is the optimal blocksize for a call of CUNMQR.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(n,m).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:n)` is stored on exit in A(i+1:n,i), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine CUNGQR.

To use Q to update another matrix, use LAPACK subroutine CUNMQR.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(n,p).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a complex scalar, and v is a complex vector with `v(p-k+i+1:p)` = 0 and `v(p-k+i)` = 1; `v(1:p-k+i-1)` is stored on exit in B(n-k+i,1:p-k+i-1), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine CUNGRQ.

To use Z to update another matrix, use LAPACK subroutine CUNMRQ.

NAME

zggrqf - compute a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B

SYNOPSIS

```
 SUBROUTINE ZGGRQF( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK, LWORK,
*      INFO)
 DOUBLE COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER M, P, N, LDA, LDB, LWORK, INFO

 SUBROUTINE ZGGRQF_64( M, P, N, A, LDA, TAUA, B, LDB, TAUB, WORK,
*      LWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), TAUA(*), B(LDB,*), TAUB(*), WORK(*)
 INTEGER*8 M, P, N, LDA, LDB, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE GGRQF( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, P, N, LDA, LDB, LWORK, INFO

 SUBROUTINE GGRQF_64( [M], [P], [N], A, [LDA], TAUA, B, [LDB], TAUB,
*       [WORK], [LWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: TAUA, TAUB, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, P, N, LDA, LDB, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zggrqf(int m, int p, int n, doublecomplex *a, int lda, doublecomplex *taua, doublecomplex *b, int ldb, doublecomplex

*taub, int *info);

void zggrqf_64(long m, long p, long n, doublecomplex *a, long lda, doublecomplex *taua, doublecomplex *b, long ldb, doublecomplex *taub, long *info);

PURPOSE

zggrqf computes a generalized RQ factorization of an M-by-N matrix A and a P-by-N matrix B:

```
A = R*Q,        B = Z*T*Q,
```

where Q is an N-by-N unitary matrix, Z is a P-by-P unitary matrix, and R and T assume one of the forms:

if M <= N, R = (0 R12) M, or if M > N, R = (R11) M-N, N-M M (R21) N N

where R12 or R21 is upper triangular, and

if P >= N, T = (T11) N , or if P < N, T = (T11 T12) P, (0) P-N P N-P N

where T11 is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of A*inv(B):

```
A*inv(B) = (R*inv(T))*Z'
```

where `inv(B)` denotes the inverse of the matrix B, and Z' denotes the conjugate transpose of the matrix Z.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **P (input)**
  The number of rows of the matrix B. P > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, if M < = N, the upper triangle of the subarray `A(1:M,N-M+1:N)` contains the M-by-M upper triangular matrix R; if M > N, the elements on and above the (M-N)-th subdiagonal contain the M-by-N upper trapezoidal matrix R; the remaining elements, with the array TAUA, represent the unitary matrix Q as a product of elementary reflectors (see Further Details).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **TAUA (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Q (see Further Details).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, the elements on and above the diagonal of the array contain the min(P,N)-by-N upper trapezoidal matrix T (T is upper triangular if P > = N); the elements below the diagonal, with the array TAUB, represent the unitary matrix Z as a product of elementary reflectors (see Further Details).

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TAUB (output)**
  The scalar factors of the elementary reflectors which represent the unitary matrix Z (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N,M,P). For optimum performance LWORK >= max(N,M,P)*max(NB1,NB2,NB3), where NB1 is the optimal blocksize for the RQ factorization of an M-by-N matrix, NB2 is the optimal blocksize for the QR factorization of a P-by-N matrix, and NB3 is the optimal blocksize for a call of CUNMRQ.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO =-i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(k), where k  = min(m,n).
```

Each `H(i)` has the form

```
H(i)  = I - taua * v * v'
```

where taua is a complex scalar, and v is a complex vector with `v(n-k+i+1:n)` = 0 and `v(n-k+i)` = 1; `v(1:n-k+i-1)` is stored on exit in A(m-k+i,1:n-k+i-1), and taua in TAUA(i).

To form Q explicitly, use LAPACK subroutine CUNGRQ.

To use Q to update another matrix, use LAPACK subroutine CUNMRQ.

The matrix Z is represented as a product of elementary reflectors

```
Z  = H(1) H(2) . . . H(k), where k  = min(p,n).
```

Each `H(i)` has the form

```
H(i)  = I - taub * v * v'
```

where taub is a complex scalar, and v is a complex vector with `v(1:i-1)` = 0 and `v(i)` = 1; `v(i+1:p)` is stored on exit in B(i+1:p,i), and taub in TAUB(i).

To form Z explicitly, use LAPACK subroutine CUNGQR.

To use Z to update another matrix, use LAPACK subroutine CUNMQR.

NAME

zggsvd - compute the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B

SYNOPSIS

```
 SUBROUTINE ZGGSVD( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B, LDB,
*      ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, WORK2, IWORK3, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER IWORK3(*)
 DOUBLE PRECISION ALPHA(*), BETA(*), WORK2(*)

 SUBROUTINE ZGGSVD_64( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B,
*      LDB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, WORK2, IWORK3,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER*8 M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK3(*)
 DOUBLE PRECISION ALPHA(*), BETA(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVD( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       [WORK2], IWORK3, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK3
 REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK2

 SUBROUTINE GGSVD_64( JOBU, JOBV, JOBQ, [M], [N], [P], K, L, A, [LDA],
*       B, [LDB], ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ], [WORK],
*       [WORK2], IWORK3, [INFO])
```

```
CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
INTEGER(8) :: M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ, INFO
INTEGER(8), DIMENSION(:) :: IWORK3
REAL(8), DIMENSION(:) :: ALPHA, BETA, WORK2
```

C INTERFACE

#include <sunperf.h>

void zggsvd(char jobu, char jobv, char jobq, int m, int n, int p, int *k, int *l, doublecomplex *a, int lda, doublecomplex *b, int ldb, double *alpha, double *beta, doublecomplex *u, int ldu, doublecomplex *v, int ldv, doublecomplex *q, int ldq, int *iwork3, int *info);

void zggsvd_64(char jobu, char jobv, char jobq, long m, long n, long p, long *k, long *l, doublecomplex *a, long lda, doublecomplex *b, long ldb, double *alpha, double *beta, doublecomplex *u, long ldu, doublecomplex *v, long ldv, doublecomplex *q, long ldq, long *iwork3, long *info);

PURPOSE

zggsvd computes the generalized singular value decomposition (GSVD) of an M-by-N complex matrix A and P-by-N complex matrix B:

```
      U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R )
```

where U, V and Q are unitary matrices, and Z' means the conjugate transpose of Z. Let K+L = the effective numerical rank of the matrix (A',B')', then R is a (K+L)-by-(K+L) nonsingular upper triangular matrix, D1 and D2 are M-by-(K+L) and P-by-(K+L) ``diagonal'' matrices and of the following structures, respectively:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
             M-K-L ( 0  0 )

                  K  L
       D2 =   L ( 0  S )
             P-L ( 0  0 )

                 N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 )
            L (  0    0   R22 )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),

  S = diag( BETA(K+1),  ... , BETA(K+L) ),

  C**2 + S**2 = I.

  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                  K M-K K+L-M

       D1 =   K ( I  0    0   )

            M-K ( 0  C    0   )

                    K M-K K+L-M

       D2 =   M-K ( 0  S    0  )

            K+L-M ( 0  0    I  )

              P-L ( 0  0    0  )

                   N-K-L  K   M-K  K+L-M

  ( 0 R ) =     K ( 0    R11  R12  R13  )

              M-K ( 0     0   R22  R23  )

            K+L-M ( 0     0    0   R33  )
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(M) ),

  S = diag( BETA(K+1),  ... , BETA(M) ),

  C**2 + S**2 = I.

  (R11 R12 R13 ) is stored in A(1:M, N-K-L+1:N), and R33 is stored
  ( 0  R22 R23 )

  in B(M-K+1:L,N+M-K-L+1:N) on exit.
```

The routine computes C, S, R, and optionally the unitary

transformation matrices U, V and Q.

In particular, if B is an N-by-N nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A*inv(B):

```
A*inv(B) = U*(D1*inv(D2))*V'.
```

If (A',B')' has orthnormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem: A'*A x = lambda* B'*B x.

In some literature, the GSVD of A and B is presented in the form U'*A*X = (0 D1), V'*B*X = (0 D2)

where U and V are orthogonal and X is nonsingular, and D1 and D2 are ``diagonal''. The former GSVD form can be converted to the latter form by taking the nonsingular matrix X as

```
X = Q*(  I   0    )

      (  0 inv(R) )
```

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Unitary matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Unitary matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Unitary matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M > = 0.
- **N (input)**
  The number of columns of the matrices A and B. N > = 0.
- **P (input)**
  The number of rows of the matrix B. P > = 0.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose. K + L = effective numerical rank of (A',B')'.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular matrix R, or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,M).
- **B (input/output)**

On entry, the P-by-N matrix B. On exit, B contains part of the triangular matrix R if M-K-L < 0. See Purpose for details.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,P).
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1, ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L > = 0, ALPHA(K+1:K+L) = C,

  BETA(K+1:K+L) = S, or if M-K-L < 0, ALPHA(K+1:M) = C, ALPHA(M+1:K+L) = 0

  BETA(K+1:M) = S, BETA(M+1:K+L) = 1 and ALPHA(K+L+1:N) = 0

  BETA(K+L+1:N) = 0
- **BETA (output)**
  See description of ALPHA.
- **U (output)**
  If JOBU = 'U', U contains the M-by-M unitary matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU > = max(1,M) if JOBU = 'U'; LDU > = 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the P-by-P unitary matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV > = max(1,P) if JOBV = 'V'; LDV > = 1 otherwise.
- **Q (output)**
  If JOBQ = 'Q', Q contains the N-by-N unitary matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N) if JOBQ = 'Q'; LDQ > = 1 otherwise.
- **WORK (workspace)**
  dimension(MAX(3*N,M,P)+N)
- **WORK2 (workspace)**
  dimension(2*N)
- **IWORK3 (output)**
  dimension(N) On exit, IWORK3 stores the sorting information. More precisely, the following loop will sort ALPHA for I = K+1, min(M,K+L) swap ALPHA(I) and ALPHA(IWORK3(I)) endfor such that ALPHA(1) > = ALPHA(2) > = ... > = ALPHA(N).
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = 1, the Jacobi-type procedure failed to
converge.  For further details, see subroutine CTGSJA.
```

NAME

zggsvp - compute unitary matrices U, V and Q such that N-K-L K L U'*A*Q = K (0 A12 A13) if M-K-L >= 0

SYNOPSIS

```
 SUBROUTINE ZGGSVP( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB, TOLA,
*      TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, RWORK, TAU, WORK,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
 INTEGER M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZGGSVP_64( JOBU, JOBV, JOBQ, M, P, N, A, LDA, B, LDB,
*      TOLA, TOLB, K, L, U, LDU, V, LDV, Q, LDQ, IWORK, RWORK, TAU,
*      WORK, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), TAU(*), WORK(*)
 INTEGER*8 M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE GGSVP( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B, [LDB],
*       TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK], [RWORK],
*       [TAU], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: RWORK
```

```
 SUBROUTINE GGSVP_64( JOBU, JOBV, JOBQ, [M], [P], [N], A, [LDA], B,
*       [LDB], TOLA, TOLB, K, L, U, [LDU], V, [LDV], Q, [LDQ], [IWORK],
*       [RWORK], [TAU], [WORK], [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER(8) :: M, P, N, LDA, LDB, K, L, LDU, LDV, LDQ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zggsvp(char jobu, char jobv, char jobq, int m, int p, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, double tola, double tolb, int *k, int *l, doublecomplex *u, int ldu, doublecomplex *v, int ldv, doublecomplex *q, int ldq, int *info);

void zggsvp_64(char jobu, char jobv, char jobq, long m, long p, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, double tola, double tolb, long *k, long *l, doublecomplex *u, long ldu, doublecomplex *v, long ldv, doublecomplex *q, long ldq, long *info);

PURPOSE

zggsvp computes unitary matrices U, V and Q such that L (0 0 A23)

```
          M-K-L ( 0     0    0  )

                   N-K-L  K    L

        =     K ( 0    A12  A13 )  if M-K-L < 0;

            M-K ( 0     0   A23 )

                 N-K-L  K    L

 V'*B*Q =   L ( 0     0   B13 )

          P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal. K+L = the effective numerical rank of the (M+P)-by-N matrix (A',B')'. Z' denotes the conjugate transpose of Z.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine CGGSVD.

ARGUMENTS

- **JOBU (input)**

  ```
  = 'U':  Unitary matrix U is computed;

  = 'N':  U is not computed.
  ```

- **JOBV (input)**

  ```
  = 'V':  Unitary matrix V is computed;

  = 'N':  V is not computed.
  ```

- **JOBQ (input)**

  ```
  = 'Q':  Unitary matrix Q is computed;

  = 'N':  Q is not computed.
  ```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A contains the triangular (or trapezoidal) matrix described in the Purpose section.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, B contains the triangular matrix described in the Purpose section.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the thresholds to determine the effective numerical rank of matrix B and a subblock of A. Generally, they are set to TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS. The size of TOLA and TOLB may affect the size of backward errors of the decomposition.
- **TOLB (input)**
  See description of TOLA.
- **K (output)**
  On exit, K and L specify the dimension of the subblocks described in Purpose section. K + L = effective numerical rank of (A',B')'.
- **L (output)**
  See the description of K.
- **U (output)**
  If JOBU = 'U', U contains the unitary matrix U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= `max(1,M)` if JOBU = 'U'; LDU >= 1 otherwise.
- **V (output)**
  If JOBV = 'V', V contains the unitary matrix V. If JOBV = 'N', V is not referenced.
- **LDV (input)**

The leading dimension of the array V. LDV >= `max(1,P)` if JOBV = 'V'; LDV >= 1 otherwise.

- **Q (output)**
  If JOBQ = 'Q', Q contains the unitary matrix Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= `max(1,N)` if JOBQ = 'Q'; LDQ >= 1 otherwise.
- **IWORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(2*N)`
- **TAU (workspace)**
  `dimension(N)`
- **WORK (workspace)**
  `dimension(MAX(3*N,M,P))`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

FURTHER DETAILS

The subroutine uses LAPACK subroutine CGEQPF for the QR factorization with column pivoting to detect the effective numerical rank of the a matrix. It may be replaced by a better rank determination strategy.

NAME

zgssco - General sparse solver condition number estimate.

SYNOPSIS

```
SUBROUTINE ZGSSCO ( COND, HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  COND
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSCO - Condition number estimate.

PARAMETERS

COND - DOUBLE PRECISION

On exit, an estimate of the condition number of the factored matrix. Must be called after the numerical factorization subroutine, ZGSSFA().

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-700 : Invalid calling sequence - need to call ZGSSFA first.
-710 : Condition number estimate not available (not implemented
       for this HANDLE's matix type).
```

NAME

zgssda - Deallocate working storage for the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSDA ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSDA - Deallocate dynamically allocated working storage.

PARAMETERS

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

none

NAME

zgssfa - General sparse solver numeric factorization.

SYNOPSIS

```
SUBROUTINE ZGSSFA ( NEQNS, COLSTR, ROWIND, VALUES, HANDLE, IER )

INTEGER           NEQNS, COLSTR(*), ROWIND(*), IER
DOUBLE COMPLEX    VALUES(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSFA - Numeric factorization of a sparse matrix.

PARAMETERS

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

`COLSTR(*)` - INTEGER array

On entry, `COLSTR(*)` is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

`ROWIND(*)` - INTEGER array

On entry, `ROWIND(*)` is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

`VALUES(*)` - DOUBLE COMPLEX array

On entry, `VALUES(*)` is an array of size COLSTR(NEQNS+1)-1, containing the numeric values of the sparse matrix to be factored. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-300 : Invalid calling sequence - need to call ZGSSOR first.
-301 : Failure to dynamically allocate memory.
-666 : Internal error.
```

NAME

zgssfs - General sparse solver one call interface.

SYNOPSIS

```
SUBROUTINE ZGSSFS ( MTXTYP, PIVOT , NEQNS, COLSTR, ROWIND,
                    VALUES, NRHS  , RHS   , LDRHS , ORDMTHD,
                    OUTUNT, MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), NRHS, LDRHS,
                  OUTUNT, MSGLVL, IER
CHARACTER*3       ORDMTHD
DOUBLE COMPLEX    VALUES(*), RHS(*)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSFS - General sparse solver one call interface.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, pivot specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

VALUES(*) - DOUBLE COMPLEX array

On entry, VALUES(*) is an array of size COLSTR(NEQNS+1)-1, containing the non-zero numeric values of the sparse matrix to be factored. Unchanged on exit.

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

RHS(*) - DOUBLE COMPLEX array

On entry, RHS(LDRHS,NRHS) contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see ZGSSUO)
```

Unchanged on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array of containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros is less than NEQNS.
-201 : Failure to dynamically allocate memory.
-301 : Failure to dynamically allocate memory.
-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS
-666 : Internal error.

NAME

zgssin - Initialize the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSIN ( MTXTYP, PIVOT, NEQNS, COLSTR, ROWIND, OUTUNT,
                    MSGLVL, HANDLE, IER )

CHARACTER*2       MTXTYP
CHARACTER*1       PIVOT
INTEGER           NEQNS, COLSTR(*), ROWIND(*), OUTUNT, MSGLVL, IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSIN - Initialize the sparse solver and input the matrix structure.

PARAMETERS

MTXTYP - CHARACTER*2

On entry, MTXTYP specifies the coefficient matrix type. Specifically, the valid options are:

```
'ss' or 'SS' - symmetric structure, symmetric values
'su' or 'SU' - symmetric structure, unsymmetric values
'uu' or 'UU' - unsymmetric structure, unsymmetric values
```

Unchanged on exit.

PIVOT - CHARACTER*1

On entry, PIVOT specifies whether or not pivoting is used in the course of the numeric factorization. The valid options are:

```
'n' or 'N' - no pivoting is used
(Pivoting is not supported for this release).
```

Unchanged on exit.

NEQNS - INTEGER

On entry, NEQNS specifies the number of equations in coefficient matrix. Unchanged on exit.

COLSTR(*) - INTEGER array

On entry, COLSTR(*) is an array of size (NEQNS+1), containing the pointers of the matrix structure. Unchanged on exit.

ROWIND(*) - INTEGER array

On entry, ROWIND(*) is an array of size COLSTR(NEQNS+1)-1, containing the indices of the matrix structure. Unchanged on exit.

HANDLE(150) - DOUBLE PRECISION array

On entry, HANDLE(*) is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

OUTUNT - INTEGER

Output unit. Unchanged on exit.

MSGLVL - INTEGER

Message level.

```
0 - no output from solver.
(No messages supported for this release.)
```

Unchanged on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-101 : Failure to dynamically allocate memory.
-102 : Invalid matrix type.
-103 : Invalid pivot option.
-104 : Number of nonzeros less than NEQNS.
```

NAME

zgssor - General sparse solver ordering and symbolic factorization.

SYNOPSIS

```
SUBROUTINE ZGSSOR ( ORDMTHD, HANDLE, IER )

CHARACTER*3       ORDMTHD
INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSOR - Orders and symbolically factors a sparse matrix.

PARAMETERS

ORDMTHD - CHARACTER*3

On entry, ORDMTHD specifies the fill-reducing ordering to be used by the sparse solver. Specifically, the valid options are:

```
'nat' or 'NAT' - natural ordering (no ordering)
'mmd' or 'MMD' - multiple minimum degree
'gnd' or 'GND' - general nested dissection
'uso' or 'USO' - user specified ordering (see ZGSSUO)
```

Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

-200 : Invalid calling sequence - need to call ZGSSIN first.
-201 : Failure to dynamically allocate memory.
-666 : Internal error.

NAME

zgssps - Print general sparse solver statics.

SYNOPSIS

```
SUBROUTINE ZGSSPS ( HANDLE, IER )

INTEGER           IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSPS - Print solver statistics.

PARAMETERS

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-800 : Invalid calling sequence - need to call ZGSSSL first.
-899 : Printed solver statistics not supported this release.
```

NAME

zgssrp - Return permutation used by the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSRP ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSRP - Returns the permutation used by the solver for the fill-reducing ordering.

PARAMETERS

`PERM(NEQNS)` - INTEGER array

Undefined on entry. `PERM(NEQNS)` is the permutation array used by the sparse solver for the fill-reducing ordering. Modified on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-600 : Invalid calling sequence - need to call ZGSSOR first.
```

NAME

zgsssl - Solve routine for the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSSL ( NRHS, RHS, LDRHS, HANDLE, IER )

INTEGER           NRHS, LDRHS, IER
DOUBLE COMPLEX    RHS(LDRHS,NRHS)
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSSL - Triangular solve of a factored sparse matrix.

PARAMETERS

NRHS - INTEGER

On entry, NRHS specifies the number of right hand sides to solve for. Unchanged on exit.

`RHS(LDRHS,*)` - DOUBLE COMPLEX array

On entry, `RHS(LDRHS,NRHS)` contains the NRHS right hand sides. On exit, it contains the solutions.

LDRHS - INTEGER

On entry, LDRHS specifies the leading dimension of the RHS array. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-400 : Invalid calling sequence - need to call ZGSSFA first.
```

-401 : Failure to dynamically allocate memory.
-402 : NRHS < 1
-403 : NEQNS > LDRHS

NAME

zgssuo - User supplied permutation for ordering used in the general sparse solver.

SYNOPSIS

```
SUBROUTINE ZGSSUO ( PERM, HANDLE, IER )

INTEGER           PERM(*), IER
DOUBLE PRECISION  HANDLE(150)
```

PURPOSE

ZGSSUO - User supplied permutation for ordering. Must be called after `ZGSSIN()` (sparse solver initialization) and before `ZGSSOR()` (sparse solver ordering).

PARAMETERS

`PERM(NEQNS)` - INTEGER array

On entry, `PERM(NEQNS)` is a permutation array supplied by the user for the fill-reducing ordering. Unchanged on exit.

`HANDLE(150)` - DOUBLE PRECISION array

On entry, `HANDLE(*)` is an array containing information needed by the solver, and must be passed unchanged to each sparse solver subroutine. Modified on exit.

IER - INTEGER

Error number. If no error encountered, unchanged on exit. If error encountered, it is set to a non-zero integer. Error numbers set by this subroutine:

```
-500 : Invalid calling sequence - need to call ZGSSIN first.
```

NAME

zgtcon - estimate the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF

SYNOPSIS

```
 SUBROUTINE ZGTCON( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
 INTEGER N, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND

 SUBROUTINE ZGTCON_64( NORM, N, LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*       RCOND, WORK, INFO)
 CHARACTER * 1 NORM
 DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), WORK(*)
 INTEGER*8 N, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE GTCON( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
 INTEGER :: N, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: ANORM, RCOND

 SUBROUTINE GTCON_64( NORM, [N], LOW, DIAG, UP1, UP2, IPIVOT, ANORM,
*        RCOND, [WORK], [INFO])
 CHARACTER(LEN=1) :: NORM
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2, WORK
 INTEGER(8) :: N, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

```
REAL(8) :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void zgtcon(char norm, int n, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, int *ipivot, double anorm, double *rcond, int *info);

void zgtcon_64(char norm, long n, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zgtcon estimates the reciprocal of the condition number of a complex tridiagonal matrix A using the LU factorization as computed by CGTTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by CGTTRF.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **ANORM (input)**
  If NORM = '1' or 'O', the 1-norm of the original matrix A. If NORM = 'I', the infinity-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.

- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

zgthr - Gathers specified elements from y into x.

SYNOPSIS

```
SUBROUTINE ZGTHR(NZ, Y, X, INDX)

DOUBLE COMPLEX Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE ZGTHR_64(NZ, Y, X, INDX)

DOUBLE COMPLEX Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHR([NZ], Y, X, INDX)

```
COMPLEX(8), DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHR_64([NZ], Y, X, INDX)

COMPLEX(8), DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZGTHR - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input)

Vector in full storage form. Unchanged on exit.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

zgthrz - Gather and zero.

SYNOPSIS

```
SUBROUTINE ZGTHRZ(NZ, Y, X, INDX)

DOUBLE COMPLEX Y(*), X(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE ZGTHRZ_64(NZ, Y, X, INDX)

DOUBLE COMPLEX Y(*), X(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE GTHRZ([NZ], Y, X, INDX)

```
COMPLEX(8), DIMENSION(:) :: Y, X
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE GTHRZ_64([NZ], Y, X, INDX)

COMPLEX(8), DIMENSION(:) :: Y, X
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZGTHRZ - Gathers the specified elements from a vector y in full storage form into a vector x in compressed form. The gathered elements of y are set to zero. Only the elements of y whose indices are listed in indx are referenced.

```
do i = 1, n
  x(i) = y(indx(i))
  y(indx(i)) = 0
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

Y (input/output)

Vector in full storage form. Gathered elements are set to zero.

X (output)

Vector in compressed form. Contains elements of y whose indices are listed in indx on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

NAME

zgtrfs - improve the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZGTRFS( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGTRFS_64( TRANSA, N, NRHS, LOW, DIAG, UP, LOWF, DIAGF,
*      UPF1, UPF2, IPIVOT, B, LDB, X, LDX, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GTRFS( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF, DIAGF,
*       UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GTRFS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgtrfs(char transa, int n, int nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *lowf, doublecomplex *diagf, doublecomplex *upf1, doublecomplex *upf2, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zgtrfs_64(char transa, long n, long nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *lowf, doublecomplex *diagf, doublecomplex *upf1, doublecomplex *upf2, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zgtrfs improves the computed solution to a system of linear equations when the coefficient matrix is tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose)
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The diagonal elements of A.
- **UP (input)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A as computed by CGTTRF.
- **DIAGF (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UPF1 (input)**
  The (n-1) elements of the first superdiagonal of U.
- **UPF2 (input)**
  The (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CGTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude

of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zgtsv - solve the equation A*X = B,

SYNOPSIS

```
SUBROUTINE ZGTSV( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE ZGTSV_64( N, NRHS, LOW, DIAG, UP, B, LDB, INFO)
DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE GTSV( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE GTSV_64( [N], [NRHS], LOW, DIAG, UP, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zgtsv(int n, int nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *b, int ldb, int *info);

void zgtsv_64(long n, long nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *b, long ldb, long *info);

PURPOSE

zgtsv solves the equation

where A is an N-by-N tridiagonal matrix, by Gaussian elimination with partial pivoting.

Note that the equation A'*X = B may be solved by interchanging the order of the arguments DU and DL.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) subdiagonal elements of A. On exit, LOW is overwritten by the (n-2) elements of the second superdiagonal of the upper triangular matrix U from the LU factorization of A, in LOW(1), ..., LOW(n-2).
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A. On exit, DIAG is overwritten by the n diagonal elements of U.
- **UP (input/output)**
  On entry, UP must contain the (n-1) superdiagonal elements of A. On exit, UP is overwritten by the (n-1) elements of the first superdiagonal of U.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, U(i,i) is exactly zero, and the solution
has not been computed.  The factorization has not been
completed unless i  = N.
```

NAME

zgtsvx - use the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZGTSVX( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZGTSVX_64( FACT, TRANSA, N, NRHS, LOW, DIAG, UP, LOWF,
*      DIAGF, UPF1, UPF2, IPIVOT, B, LDB, X, LDX, RCOND, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 FACT, TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP(*), LOWF(*), DIAGF(*), UPF1(*), UPF2(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE GTSVX( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP, LOWF,
*       DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE GTSVX_64( FACT, [TRANSA], [N], [NRHS], LOW, DIAG, UP,
*       LOWF, DIAGF, UPF1, UPF2, IPIVOT, B, [LDB], X, [LDX], RCOND, FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zgtsvx(char fact, char transa, int n, int nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *lowf, doublecomplex *diagf, doublecomplex *upf1, doublecomplex *upf2, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zgtsvx_64(char fact, char transa, long n, long nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up, doublecomplex *lowf, doublecomplex *diagf, doublecomplex *upf1, doublecomplex *upf2, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zgtsvx uses the LU factorization to compute the solution to a complex system of linear equations A * X = B, A**T * X = B, or A**H * X = B, where A is a tridiagonal matrix of order N and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the LU decomposition is used to factor the matrix A as A = L * U, where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

2. If some U(i,i)=0, so that U is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': LOWF, DIAGF, UPF1, UPF2, and IPIVOT contain the factored form of A; LOW, DIAG, UP, LOWF, DIAGF, UPF1, UPF2 and IPIVOT will not be modified. = 'N': The matrix will be copied to LOWF, DIAGF, and UPF1 and factored.
- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) subdiagonal elements of A.
- **DIAG (input)**
  The n diagonal elements of A.
- **UP (input/output)**
  The (n-1) superdiagonal elements of A.
- **LOWF (input/output)**
  If FACT = 'F', then LOWF is an input argument and on entry contains the (n-1) multipliers that define the matrix L from the LU factorization of A as computed by CGTTRF.

  If FACT = 'N', then LOWF is an output argument and on exit contains the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAGF (input/output)**

If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

- **UPF1 (input/output)**
  If FACT = 'F', then UPF1 is an input argument and on entry contains the (n-1) elements of the first superdiagonal of U.

  If FACT = 'N', then UPF1 is an output argument and on exit contains the (n-1) elements of the first superdiagonal of U.
- **UPF2 (input/output)**
  If FACT = 'F', then UPF2 is an input argument and on entry contains the (n-2) elements of the second superdiagonal of U.

  If FACT = 'N', then UPF2 is an output argument and on exit contains the (n-2) elements of the second superdiagonal of U.
- **IPIVOT (input/output)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains the pivot indices from the LU factorization of A as computed by CGTTRF.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains the pivot indices from the LU factorization of A; row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  U(i,i) is exactly zero.  The factorization
has not been completed unless i  = N, but the
factor U is exactly singular, so the solution
and error bounds could not be computed.
RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zgttrf - compute an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges

---

SYNOPSIS

```
SUBROUTINE ZGTTRF( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZGTTRF_64( N, LOW, DIAG, UP1, UP2, IPIVOT, INFO)
DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE GTTRF( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE GTTRF_64( [N], LOW, DIAG, UP1, UP2, IPIVOT, [INFO])
COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgttrf(int n, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, int *ipivot, int *info);

void zgttrf_64(long n, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, long *ipivot,

long *info);

PURPOSE

zgttrf computes an LU factorization of a complex tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

```
A = L * U
```

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

ARGUMENTS

- **N (input)**
  The order of the matrix A.
- **LOW (input/output)**
  On entry, LOW must contain the (n-1) sub-diagonal elements of A.

  On exit, LOW is overwritten by the (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input/output)**
  On entry, DIAG must contain the diagonal elements of A.

  On exit, DIAG is overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input/output)**
  On entry, UP1 must contain the (n-1) super-diagonal elements of A.

  On exit, UP1 is overwritten by the (n-1) elements of the first super-diagonal of U.
- **UP2 (output)**
  On exit, UP2 is overwritten by the (n-2) elements of the second super-diagonal of U.
- **IPIVOT (output)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **INFO (output)**

  ```
   = 0:  successful exit

   < 0:  if INFO  = -k, the k-th argument had an illegal value

   > 0:  if INFO  = k, U(k,k) is exactly zero. The factorization
  has been completed, but the factor U is exactly
  singular, and division by zero will occur if it is used
  to solve a system of equations.
  ```

NAME

zgttrs - solve one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZGTTRS( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT, B,
*      LDB, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
 INTEGER N, NRHS, LDB, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE ZGTTRS_64( TRANSA, N, NRHS, LOW, DIAG, UP1, UP2, IPIVOT,
*      B, LDB, INFO)
 CHARACTER * 1 TRANSA
 DOUBLE COMPLEX LOW(*), DIAG(*), UP1(*), UP2(*), B(LDB,*)
 INTEGER*8 N, NRHS, LDB, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE GTTRS( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
 COMPLEX(8), DIMENSION(:,:) :: B
 INTEGER :: N, NRHS, LDB, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE GTTRS_64( [TRANSA], [N], [NRHS], LOW, DIAG, UP1, UP2,
*       IPIVOT, B, [LDB], [INFO])
 CHARACTER(LEN=1) :: TRANSA
 COMPLEX(8), DIMENSION(:) :: LOW, DIAG, UP1, UP2
 COMPLEX(8), DIMENSION(:,:) :: B
 INTEGER(8) :: N, NRHS, LDB, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zgttrs(char transa, int n, int nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, int *ipivot, doublecomplex *b, int ldb, int *info);

void zgttrs_64(char transa, long n, long nrhs, doublecomplex *low, doublecomplex *diag, doublecomplex *up1, doublecomplex *up2, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zgttrs solves one of the systems of equations A * X = B, A**T * X = B, or A**H * X = B, with a tridiagonal matrix A using the LU factorization computed by CGTTRF.

ARGUMENTS

- **TRANSA (input)**
  Specifies the form of the system of equations. = 'N': A * X = B (No transpose)

  ```
  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```
- **N (input)**
  The order of the matrix A.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **LOW (input)**
  The (n-1) multipliers that define the matrix L from the LU factorization of A.
- **DIAG (input)**
  The n diagonal elements of the upper triangular matrix U from the LU factorization of A.
- **UP1 (input)**
  The (n-1) elements of the first super-diagonal of U.
- **UP2 (input)**
  The (n-2) elements of the second super-diagonal of U.
- **IPIVOT (input)**
  The pivot indices; for 1 < = i < = n, row i of the matrix was interchanged with row IPIVOT(i). `IPIVOT(i)` will always be either i or i+1; `IPIVOT(i)` = i indicates a row interchange was not required.
- **B (input/output)**
  On entry, the matrix of right hand side vectors B. On exit, B is overwritten by the solution vectors X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

zhbev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE ZHBEV( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, NDIAG, LDA, LDZ, INFO
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHBEV_64( JOBZ, UPLO, N, NDIAG, A, LDA, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, LDZ, INFO
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEV( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER :: N, NDIAG, LDA, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK2

 SUBROUTINE HBEV_64( JOBZ, UPLO, [N], NDIAG, A, [LDA], W, Z, [LDZ],
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, NDIAG, LDA, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhbev(char jobz, char uplo, int n, int ndiag, doublecomplex *a, int lda, double *w, doublecomplex *z, int ldz, int *info);

void zhbev_64(char jobz, char uplo, long n, long ndiag, doublecomplex *a, long lda, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhbev computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows NDIAG and NDIAG+1 of A, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG + 1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**

The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

zhbevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE ZHBEVD( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX WORK(*)
 DOUBLE COMPLEX AB(LDAB,*), Z(LDZ,*)
 INTEGER N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 REAL RWORK(*)
 DOUBLE PRECISION W(*)

 SUBROUTINE ZHBEVD_64( JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 COMPLEX WORK(*)
 DOUBLE COMPLEX AB(LDAB,*), Z(LDZ,*)
 INTEGER*8 N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 REAL RWORK(*)
 DOUBLE PRECISION W(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEVD( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
*       WORK, [LWORK], RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Z
 INTEGER :: N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: RWORK
 REAL(8), DIMENSION(:) :: W

 SUBROUTINE HBEVD_64( JOBZ, UPLO, [N], KD, AB, [LDAB], W, Z, [LDZ],
```

```
*       WORK, [LWORK], RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX, DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Z
 INTEGER(8) :: N, KD, LDAB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL, DIMENSION(:) :: RWORK
 REAL(8), DIMENSION(:) :: W
```

C INTERFACE

#include <sunperf.h>

void zhbevd(char jobz, char uplo, int n, int kd, doublecomplex *ab, int ldab, double *w, doublecomplex *z, int ldz, complex *work, int lwork, float *rwork, int lrwork, int *info);

void zhbevd_64(char jobz, char uplo, long n, long kd, doublecomplex *ab, long ldab, double *w, doublecomplex *z, long ldz, complex *work, long lwork, float *rwork, long lrwork, long *info);

PURPOSE

zhbevd computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The

j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', AB(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', AB(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd).

On exit, AB is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows KD and KD+1 of AB, and if UPLO = 'L', the diagonal and first subdiagonal of T are returned in the first two rows of AB.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD + 1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, RWORK(1) returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N .

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

zhbevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A

SYNOPSIS

```
 SUBROUTINE ZHBEVX( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ, VL,
*      VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHBEVX_64( JOBZ, RANGE, UPLO, N, NDIAG, A, LDA, Q, LDQ,
*      VL, VU, IL, IU, ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HBEVX( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q, [LDQ],
*       VL, VU, IL, IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [WORK2],
*       [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Q, Z
 INTEGER :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK2
```

```
 SUBROUTINE HBEVX_64( JOBZ, RANGE, UPLO, [N], NDIAG, A, [LDA], Q,
*       [LDQ], VL, VU, IL, IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK],
*       [WORK2], [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Q, Z
 INTEGER(8) :: N, NDIAG, LDA, LDQ, IL, IU, NFOUND, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhbevx(char jobz, char range, char uplo, int n, int ndiag, doublecomplex *a, int lda, doublecomplex *q, int ldq, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zhbevx_64(char jobz, char range, char uplo, long n, long ndiag, doublecomplex *a, long lda, doublecomplex *q, long ldq, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zhbevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
  will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**

The order of the matrix A. N >= 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, A is overwritten by values generated during the reduction to tridiagonal form.
- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG + 1.
- **Q (output)**
  If JOBZ = 'V', the N-by-N unitary matrix used in the reduction to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'V', then LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 <= NFOUND <= N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  The first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact

value of NFOUND is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

zhbgst - reduce a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y,

---

SYNOPSIS

```
 SUBROUTINE ZHBGST( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*      LDX, WORK, RWORK, INFO)
 CHARACTER * 1 VECT, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDX, INFO
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZHBGST_64( VECT, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, X,
*      LDX, WORK, RWORK, INFO)
 CHARACTER * 1 VECT, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDX, INFO
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGST( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       X, [LDX], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, X
 INTEGER :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE HBGST_64( VECT, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       X, [LDX], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, X
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDX, INFO
 REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zhbgst(char vect, char uplo, int n, int ka, int kb, doublecomplex *ab, int ldab, doublecomplex *bb, int ldbb, doublecomplex *x, int ldx, int *info);

void zhbgst_64(char vect, char uplo, long n, long ka, long kb, doublecomplex *ab, long ldab, doublecomplex *bb, long ldbb, doublecomplex *x, long ldx, long *info);

PURPOSE

zhbgst reduces a complex Hermitian-definite banded generalized eigenproblem A*x = lambda*B*x to standard form C*y = lambda*y, such that C has the same bandwidth as A.

B must have been previously factorized as S**H*S by CPBSTF, using a split Cholesky factorization. A is overwritten by C = X**H*A*X, where X = S**(-1)*Q and Q is a unitary matrix chosen to preserve the bandwidth of A.

ARGUMENTS

- **VECT (input)**

  = 'N': do not form the transformation matrix X;

  = 'V': form X.
- **UPLO (input)**

  = 'U': Upper triangle of A is stored;

  = 'L': Lower triangle of A is stored.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= KB >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the transformed matrix X**H*A*X, stored in the same format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.

- **BB (input)**
  The banded factor S from the split Cholesky factorization of B, as returned by CPBSTF, stored in the first kb+1 rows of the array.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **X (output)**
  If VECT = 'V', the n-by-n matrix X. If VECT = 'N', the array X is not referenced.
- **LDX (input)**
  The leading dimension of the array X. LDX > = `max(1,N)` if VECT = 'V'; LDX > = 1 otherwise.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

zhbgv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE ZHBGV( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*       LDZ, WORK, RWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, INFO
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHBGV_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*       Z, LDZ, WORK, RWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, INFO
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGV( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB], W,
*        Z, [LDZ], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGV_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*        W, Z, [LDZ], [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhbgv(char jobz, char uplo, int n, int ka, int kb, doublecomplex *ab, int ldab, doublecomplex *bb, int ldbb, double *w, doublecomplex *z, int ldz, int *info);

void zhbgv_64(char jobz, char uplo, long n, long ka, long kb, doublecomplex *ab, long ldab, doublecomplex *bb, long ldbb, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhbgv computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > = 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The

j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` < =i < =j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j < =i < =min(n,j+kb).

On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB > = KB+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(3*N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

NAME

zhbgvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE ZHBGVD( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W, Z,
*      LDZ, WORK, LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHBGVD_64( JOBZ, UPLO, N, KA, KB, AB, LDAB, BB, LDBB, W,
*      Z, LDZ, WORK, LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGVD( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGVD_64( JOBZ, UPLO, [N], KA, KB, AB, [LDAB], BB, [LDBB],
*       W, Z, [LDZ], [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK],
```

```
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhbgvd(char jobz, char uplo, int n, int ka, int kb, doublecomplex *ab, int ldab, doublecomplex *bb, int ldbb, double *w, doublecomplex *z, int ldz, int *info);

void zhbgvd_64(char jobz, char uplo, long n, long ka, long kb, doublecomplex *ab, long ldab, doublecomplex *bb, long ldbb, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhbgvd computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA > = 0.

- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB > =

0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(ka+1+i-j,j)` = `A(i,j)` for `max(1,j-ka)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.

- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', `BB(kb+1+i-j,j)` = `B(i,j)` for `max(1,j-kb)` <=i <=j; if UPLO = 'L', `BB(1+i-j,j)` = `B(i,j)` for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.

- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= N.

- **WORK (workspace)**
  On exit, if INFO =0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= N. If JOBZ = 'V' and N > 1, LWORK >= 2*N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **RWORK (workspace)**
  On exit, if INFO =0, `RWORK(1)` returns the optimal LRWORK.

- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK >= 1. If JOBZ = 'N' and N > 1, LRWORK >= N. If JOBZ = 'V' and N > 1, LRWORK >= 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.

- **IWORK (workspace)**
  On exit, if INFO =0, `IWORK(1)` returns the optimal LIWORK.

- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  the algorithm failed to converge:
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

zhbgvx - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x

---

SYNOPSIS

```
 SUBROUTINE ZHBGVX( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB, LDBB,
*      Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHBGVX_64( JOBZ, RANGE, UPLO, N, KA, KB, AB, LDAB, BB,
*      LDBB, Q, LDQ, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK,
*      IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX AB(LDAB,*), BB(LDBB,*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HBGVX( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Q, Z
 INTEGER :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
```

```
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HBGVX_64( JOBZ, RANGE, UPLO, [N], KA, KB, AB, [LDAB], BB,
*       [LDBB], Q, [LDQ], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, BB, Q, Z
 INTEGER(8) :: N, KA, KB, LDAB, LDBB, LDQ, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhbgvx(char jobz, char range, char uplo, int n, int ka, int kb, doublecomplex *ab, int ldab, doublecomplex *bb, int ldbb, doublecomplex *q, int ldq, double vl, double vu, int il, int iu, double abstol, int *m, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zhbgvx_64(char jobz, char range, char uplo, long n, long ka, long kb, doublecomplex *ab, long ldab, doublecomplex *bb, long ldbb, doublecomplex *q, long ldq, double vl, double vu, long il, long iu, double abstol, long *m, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zhbgvx computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form A*x=(lambda)*B*x. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;

  = 'V': all eigenvalues in the half-open interval (VL,VU]
  will be found;
  = 'I': the IL-th through IU-th eigenvalues will be found.
  ```

- **UPLO (input)**

= 'U': Upper triangles of A and B are stored;

= 'L': Lower triangles of A and B are stored.

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **KA (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KA >= 0.
- **KB (input)**
  The number of superdiagonals of the matrix B if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KB >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first ka+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', AB(ka+1+i-j,j) = A(i,j) for max(1,j-ka) <=i <=j; if UPLO = 'L', AB(1+i-j,j) = A(i,j) for j <=i <=min(n,j+ka).

  On exit, the contents of AB are destroyed.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KA+1.
- **BB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix B, stored in the first kb+1 rows of the array. The j-th column of B is stored in the j-th column of the array BB as follows: if UPLO = 'U', BB(kb+1+i-j,j) = B(i,j) for max(1,j-kb) <=i <=j; if UPLO = 'L', BB(1+i-j,j) = B(i,j) for j <=i <=min(n,j+kb).

  On exit, the factor S from the split Cholesky factorization B = S**H*S, as returned by CPBSTF.
- **LDBB (input)**
  The leading dimension of the array BB. LDBB >= KB+1.
- **Q (output)**
  If JOBZ = 'V', the n-by-n matrix used in the reduction of A*x = (lambda)*B*x to standard form, i.e. C*x = (lambda)*x, and consequently C to tridiagonal form. If JOBZ = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. If JOBZ = 'N', LDQ >= 1. If JOBZ = 'V', LDQ >= max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 <= IL <= IU <= N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing AP to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold

2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized so that Z**H*B*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = N.
- **WORK (workspace)**
  `dimension(N)`
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is:

 < = N:  then i eigenvectors failed to converge.  Their
indices are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then CPBSTF
```

  returned INFO = i: B is not positive definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

zhbmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
 SUBROUTINE ZHBMV( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA, Y,
*      INCY)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER N, NDIAG, LDA, INCX, INCY

 SUBROUTINE ZHBMV_64( UPLO, N, NDIAG, ALPHA, A, LDA, X, INCX, BETA,
*      Y, INCY)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
 INTEGER*8 N, NDIAG, LDA, INCX, INCY
```

F95 INTERFACE

```
 SUBROUTINE HBMV( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX], BETA,
*       Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INCX, INCY

 SUBROUTINE HBMV_64( UPLO, [N], NDIAG, ALPHA, A, [LDA], X, [INCX],
*       BETA, Y, [INCY])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:) :: X, Y
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, NDIAG, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zhbmv(char uplo, int n, int ndiag, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *x, int incx, doublecomplex beta, doublecomplex *y, int incy);

void zhbmv_64(char uplo, long n, long ndiag, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *x, long incx, doublecomplex beta, doublecomplex *y, long incy);

PURPOSE

zhbmv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with ndiag super-diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the band matrix A is being supplied as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is being supplied.

  UPLO = 'L' or 'l' The lower triangular part of A is being supplied.

  Unchanged on exit.

- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.

- **NDIAG (input)**
  On entry, NDIAG specifies the number of super-diagonals of the matrix A. NDIAG must satisfy 0 .le. NDIAG. Unchanged on exit.

- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.

- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer the upper triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
10    CONTINUE
20  CONTINUE
```

  Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag

triangle of the array A is not referenced. The following program segment will transfer the lower triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zhbtrd - reduce a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
 SUBROUTINE ZHBTRD( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ, WORK,
*      INFO)
 CHARACTER * 1 VECT, UPLO
 DOUBLE COMPLEX AB(LDAB,*), Q(LDQ,*), WORK(*)
 INTEGER N, KD, LDAB, LDQ, INFO
 DOUBLE PRECISION D(*), E(*)

 SUBROUTINE ZHBTRD_64( VECT, UPLO, N, KD, AB, LDAB, D, E, Q, LDQ,
*      WORK, INFO)
 CHARACTER * 1 VECT, UPLO
 DOUBLE COMPLEX AB(LDAB,*), Q(LDQ,*), WORK(*)
 INTEGER*8 N, KD, LDAB, LDQ, INFO
 DOUBLE PRECISION D(*), E(*)
```

F95 INTERFACE

```
 SUBROUTINE HBTRD( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Q
 INTEGER :: N, KD, LDAB, LDQ, INFO
 REAL(8), DIMENSION(:) :: D, E

 SUBROUTINE HBTRD_64( VECT, UPLO, [N], KD, AB, [LDAB], D, E, Q, [LDQ],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: VECT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: AB, Q
 INTEGER(8) :: N, KD, LDAB, LDQ, INFO
```

```
REAL(8), DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void zhbtrd(char vect, char uplo, int n, int kd, doublecomplex *ab, int ldab, double *d, double *e, doublecomplex *q, int ldq, int *info);

void zhbtrd_64(char vect, char uplo, long n, long kd, doublecomplex *ab, long ldab, double *d, double *e, doublecomplex *q, long ldq, long *info);

PURPOSE

zhbtrd reduces a complex Hermitian band matrix A to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **VECT (input)**

  ```
  = 'N':  do not form Q;

  = 'V':  form Q;

  = 'U':  update a matrix X, by forming X*Q.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **KD (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD >= 0.

- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). On exit, the diagonal elements of AB are overwritten by the diagonal elements of the tridiagonal matrix T; if KD > 0, the elements on the first superdiagonal (if UPLO = 'U') or the first subdiagonal (if UPLO = 'L') are overwritten by the off-diagonal elements of T; the rest of AB is overwritten by values generated during the reduction.

- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.

- **D (output)**

The diagonal elements of the tridiagonal matrix T.

- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `T(i,i+1)` if UPLO = 'U'; `E(i)` = `T(i+1,i)` if UPLO = 'L'.
- **Q (input/output)**
  On entry, if VECT = 'U', then Q must contain an N-by-N matrix X; if VECT = 'N' or 'V', then Q need not be set.

  On exit: if VECT = 'V', Q contains the N-by-N unitary matrix Q; if VECT = 'U', Q contains the product X*Q; if VECT = 'N', the array Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1, and LDQ >= N if VECT = 'V' or 'U'.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Modified by Linda Kaufman, Bell Labs.

NAME

zhecon - estimate the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
 SUBROUTINE ZHECON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND

 SUBROUTINE ZHECON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE HECON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: ANORM, RCOND

 SUBROUTINE HECON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void zhecon(char uplo, int n, doublecomplex *a, int lda, int *ipivot, double anorm, double *rcond, int *info);

void zhecon_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zhecon estimates the reciprocal of the condition number of a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zheev - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE ZHEEV( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, WORK2,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, LDWORK, INFO
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHEEV_64( JOBZ, UPLO, N, A, LDA, W, WORK, LDWORK, WORK2,
*      INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, LDWORK, INFO
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEV( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK2

 SUBROUTINE HEEV_64( JOBZ, UPLO, [N], A, [LDA], W, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zheev(char jobz, char uplo, int n, doublecomplex *a, int lda, double *w, int *info);

void zheev_64(char jobz, char uplo, long n, doublecomplex *a, long lda, double *w, long *info);

PURPOSE

zheev computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,2*N-1). For optimal efficiency, LDWORK >= (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.

- **WORK2 (workspace)**

dimension(max(1,3*N-2))

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

  > 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

zheevd - compute all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE ZHEEVD( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHEEVD_64( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVD( JOBZ, UPLO, [N], A, [LDA], W, WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HEEVD_64( JOBZ, UPLO, [N], A, [LDA], W, WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
```

```
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zheevd(char jobz, char uplo, int n, doublecomplex *a, int lda, double *w, doublecomplex *work, int lwork, double *rwork, int lrwork, int *info);

void zheevd_64(char jobz, char uplo, long n, doublecomplex *a, long lda, double *w, doublecomplex *work, long lwork, double *rwork, long lrwork, long *info);

PURPOSE

zheevd computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, if JOBZ = 'V', then if INFO = 0, A contains the orthonormal eigenvectors of the matrix A. If JOBZ = 'N', then on exit the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N + 1. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, RWORK(1) returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If N < = 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK must be at least 1. If JOBZ = 'N' and N > 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

FURTHER DETAILS

Based on contributions by

```
Jeff Rutter, Computer Science Division, University of California
at Berkeley, USA
```

NAME

zheevr - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T

SYNOPSIS

```
 SUBROUTINE ZHEEVR( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, RWORK, LRWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHEEVR_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABSTOL, M, W, Z, LDZ, ISUPPZ, WORK, LWORK, RWORK, LRWORK, IWORK,
*      LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVR( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [RWORK], [LRWORK],
*       [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

```
 SUBROUTINE HEEVR_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [RWORK],
*       [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, IL, IU, M, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zheevr(char jobz, char range, char uplo, int n, doublecomplex *a, int lda, double vl, double vu, int il, int iu, double abstol, int *m, double *w, doublecomplex *z, int ldz, int *isuppz, int *info);

void zheevr_64(char jobz, char range, char uplo, long n, doublecomplex *a, long lda, double vl, double vu, long il, long iu, double abstol, long *m, double *w, doublecomplex *z, long ldz, long *isuppz, long *info);

PURPOSE

zheevr computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian tridiagonal matrix T. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, CHEEVR calls CSTEGR to compute the

eigenspectrum using Relatively Robust Representations. CSTEGR computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various ``good" L D L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of T,

```
   (a) Compute T - sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T
        is a relatively robust representation,

   (b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high
       relative accuracy by the dqds algorithm,

   (c) If there is a cluster of close eigenvalues, "choose" sigma_i
       close to the cluster, and go to step (a),

   (d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
       compute the corresponding eigenvector by forming a
       rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB//CSD-97-971, UC Berkeley, May 1997.

Note 1 : CHEEVR calls CSTEGR when the full spectrum is requested on machines which conform to the ieee-754 floating point standard. CHEEVR calls SSTEBZ and CSTEIN on non-ieee machines and

when partial spectrum requests are made.

Normal execution of CSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the ieee standard default manner.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is

determined to lie in an interval [a,b] of width less than or equal to

ABSTOL + EPS * max(|a|,|b|) ,

where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.

If high relative accuracy is important, set ABSTOL to SLAMCH('Safe minimum'). Doing so will guarantee that eigenvalues are computed to high relative accuracy when possible in future releases. The current code does not make any guarantees about high relative accuracy, but furutre releases will. See J. Barlow and J. Demmel, ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7, for a discussion of which matrices define their eigenvalues to high relative accuracy.

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,2*N). For optimal efficiency, LWORK > = (NB+1)*N, where NB is the max of the blocksize for CHETRD and for CUNMTR as returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal (and minimal) LRWORK.
- **LRWORK (input)**
  The length of the array RWORK. LRWORK > = max(1,24*N).

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal (and minimal) LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK > = max(1,10*N).

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

= 0: successful exit

< 0: if INFO = -i, the i-th argument had an illegal value

> 0: Internal error

FURTHER DETAILS

Based on contributions by

Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA

NAME

zheevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A

SYNOPSIS

```
 SUBROUTINE ZHEEVX( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHEEVX_64( JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
*      ABTOL, NFOUND, W, Z, LDZ, WORK, LDWORK, WORK2, IWORK3, IFAIL,
*      INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEEVX( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL, IU,
*       ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [LDWORK], [WORK2], [IWORK3],
*       IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK2
```

```
 SUBROUTINE HEEVX_64( JOBZ, RANGE, UPLO, [N], A, [LDA], VL, VU, IL,
*       IU, ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [LDWORK], [WORK2],
*       [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, Z
 INTEGER(8) :: N, LDA, IL, IU, NFOUND, LDZ, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zheevx(char jobz, char range, char uplo, int n, doublecomplex *a, int lda, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zheevx_64(char jobz, char range, char uplo, long n, doublecomplex *a, long lda, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zheevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A. On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  On normal exit, the first NFOUND elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK > = max(1,2*N). For optimal efficiency, LDWORK > = (NB+1)*N, where NB is the max of the blocksize for CHETRD and for CUNMTR as returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued

by XERBLA.

- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

zhegs2 - reduce a complex Hermitian-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE ZHEGS2( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER ITYPE, N, LDA, LDB, INFO

SUBROUTINE ZHEGS2_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 ITYPE, N, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE HEGS2( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: ITYPE, N, LDA, LDB, INFO

SUBROUTINE HEGS2_64( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zhegs2(int itype, char uplo, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zhegs2_64(long itype, char uplo, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zhegs2 reduces a complex Hermitian-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U')*A*inv(U)` or `inv(L)*A*inv(L')`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U` or L'*A*L.

B must have been previously factorized as U'*U or L*L' by CPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U')*A*inv(U) or inv(L)*A*inv(L');

  = 2 or 3: compute U*A*U' or L'*A*L.
  ```

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored, and how B has been factorized. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by CPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit.

  < 0:  if INFO  = -i, the i-th argument had an illegal value.
  ```

NAME

zhegst - reduce a complex Hermitian-definite generalized eigenproblem to standard form

---

SYNOPSIS

```
SUBROUTINE ZHEGST( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER ITYPE, N, LDA, LDB, INFO

SUBROUTINE ZHEGST_64( ITYPE, UPLO, N, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 ITYPE, N, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE HEGST( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: ITYPE, N, LDA, LDB, INFO

SUBROUTINE HEGST_64( ITYPE, UPLO, N, A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zhegst(int itype, char uplo, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zhegst_64(long itype, char uplo, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zhegst reduces a complex Hermitian-definite generalized eigenproblem to standard form.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**H)*A*inv(U)` or `inv(L)*A*inv(L**H)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**H or L**H*A*L.

B must have been previously factorized as U**H*U or L*L**H by CPOTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U**H)*A*inv(U) or inv(L)*A*inv(L**H);

  = 2 or 3: compute U*A*U**H or L**H*A*L.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored and B is factored as
U**H*U;
  = 'L':  Lower triangle of A is stored and B is factored as
L*L**H.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input)**
  The triangular factor from the Cholesky factorization of B, as returned by CPOTRF.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zhegv - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHEGV( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHEGV_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LDWORK, WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, LDWORK, INFO
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGV( ITYPE, JOBZ, UPLO, N, A, [LDA], B, [LDB], W, [WORK],
*       [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK2

 SUBROUTINE HEGV_64( ITYPE, JOBZ, UPLO, N, A, [LDA], B, [LDB], W,
*       [WORK], [LDWORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: ITYPE, N, LDA, LDB, LDWORK, INFO
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhegv(int itype, char jobz, char uplo, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, double *w, int *info);

void zhegv_64(long itype, char jobz, char uplo, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, double *w, long *info);

PURPOSE

zhegv computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also

positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the Hermitian positive definite matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO <= N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.
- **LDWORK (input)**
  The length of the array WORK. LDWORK >= max(1,2*N-1). For optimal efficiency, LDWORK >= (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPOTRF or CHEEV returned an error code:

 <= N:  if INFO  = i, CHEEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  <= i  <= N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

zhegvd - compute all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHEGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHEGVD_64( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGVD( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HEGVD_64( ITYPE, JOBZ, UPLO, [N], A, [LDA], B, [LDB], W,
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
```

```
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: ITYPE, N, LDA, LDB, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhegvd(int itype, char jobz, char uplo, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, double *w, int *info);

void zhegvd_64(long itype, char jobz, char uplo, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, double *w, long *info);

PURPOSE

zhegvd computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, if JOBZ = 'V', then if INFO = 0, A contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then on exit the upper triangle (if UPLO ='U') or the lower triangle (if UPLO ='L') of A, including the diagonal, is destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the Hermitian matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. If N < = 1, LWORK > = 1. If JOBZ = 'N' and N > 1, LWORK > = N + 1. If JOBZ = 'V' and N > 1, LWORK > = 2*N + N**2.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If N < = 1, LRWORK > = 1. If JOBZ = 'N' and N > 1, LRWORK > = N. If JOBZ = 'V' and N > 1, LRWORK > = 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If N < = 1, LIWORK > = 1. If JOBZ = 'N' and N > 1, LIWORK > = 1. If JOBZ = 'V' and N > 1, LIWORK > = 3 + 5*N.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  CPOTRF or CHEEVD returned an error code:
```

```
 < = N:  if INFO  = i, CHEEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not converge to zero;
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

zhegvx - compute selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHEGVX( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB, VL,
*      VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHEGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB,
*      VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK,
*      IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HEGVX( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B, [LDB],
*       VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, Z
 INTEGER :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
```

```
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HEGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], A, [LDA], B,
*       [LDB], VL, VU, IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [LWORK],
*       [RWORK], [IWORK], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, Z
 INTEGER(8) :: ITYPE, N, LDA, LDB, IL, IU, M, LDZ, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhegvx(int itype, char jobz, char range, char uplo, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, double vl, double vu, int il, int iu, double abstol, int *m, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zhegvx_64(long itype, char jobz, char range, char uplo, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, double vl, double vu, long il, long iu, double abstol, long *m, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zhegvx computes selected eigenvalues, and optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A.

  On exit, the lower triangle (if UPLO ='L') or the upper triangle (if UPLO ='U') of A, including the diagonal, is destroyed.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the Hermitian matrix B. If UPLO = 'U', the leading N-by-N upper triangular part of B contains the upper triangular part of the matrix B. If UPLO = 'L', the leading N-by-N lower triangular part of B contains the lower triangular part of the matrix B.

  On exit, if INFO < = N, the part of B containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H.

- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.

- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.

- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.

- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.

- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').

- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**T*B*Z = I; if ITYPE = 3, Z**T*inv(B)*Z = I.

  If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The length of the array WORK. LWORK > = max(1,2*N-1). For optimal efficiency, LWORK > = (NB+1)*N, where NB is the blocksize for CHETRD returned by ILAENV.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPOTRF or CHEEVX returned an error code:

 < = N:  if INFO  = i, CHEEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = N, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA

NAME

zhemm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZHEMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, LDA, LDB, LDC

 SUBROUTINE ZHEMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE HEMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, LDA, LDB, LDC

 SUBROUTINE HEMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void zhemm(char side, char uplo, int m, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex beta, doublecomplex *c, int ldc);

void zhemm_64(char side, char uplo, long m, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex beta, doublecomplex *c, long ldc);

PURPOSE

zhemm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the hermitian matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the hermitian matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the hermitian matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the hermitian matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the hermitian matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the hermitian matrix, such that when

UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced.

Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB must be at least max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry.

  On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, m). Unchanged on exit.

NAME

zhemv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE ZHEMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
INTEGER N, LDA, INCX, INCY

SUBROUTINE ZHEMV_64( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX A(LDA,*), X(*), Y(*)
INTEGER*8 N, LDA, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE HEMV( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCX, INCY

SUBROUTINE HEMV_64( UPLO, [N], ALPHA, A, [LDA], X, [INCX], BETA, Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zhemv(char uplo, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *x, int incx, doublecomplex beta, doublecomplex *y, int incy);

void zhemv_64(char uplo, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *x, long incx, doublecomplex beta, doublecomplex *y, long incy);

PURPOSE

zhemv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y

is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

zher - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

SYNOPSIS

```
SUBROUTINE ZHER( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
DOUBLE COMPLEX X(*), A(LDA,*)
INTEGER N, INCX, LDA
DOUBLE PRECISION ALPHA

SUBROUTINE ZHER_64( UPLO, N, ALPHA, X, INCX, A, LDA)
CHARACTER * 1 UPLO
DOUBLE COMPLEX X(*), A(LDA,*)
INTEGER*8 N, INCX, LDA
DOUBLE PRECISION ALPHA
```

F95 INTERFACE

```
SUBROUTINE HER( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: X
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, INCX, LDA
REAL(8) :: ALPHA

SUBROUTINE HER_64( UPLO, [N], ALPHA, X, [INCX], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: X
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, INCX, LDA
REAL(8) :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void zher(char uplo, int n, double alpha, doublecomplex *x, int incx, doublecomplex *a, int lda);

void zher_64(char uplo, long n, double alpha, doublecomplex *x, long incx, doublecomplex *a, long lda);

PURPOSE

zher performs the hermitian rank 1 operation A := alpha*x*conjg(x') + A where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.

NAME

zher2 - perform the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

SYNOPSIS

```
SUBROUTINE ZHER2( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER N, INCX, INCY, LDA

SUBROUTINE ZHER2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(LDA,*)
INTEGER*8 N, INCX, INCY, LDA
```

F95 INTERFACE

```
SUBROUTINE HER2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, INCX, INCY, LDA

SUBROUTINE HER2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A, [LDA])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, INCX, INCY, LDA
```

C INTERFACE

#include <sunperf.h>

void zher2(char uplo, int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy, doublecomplex *a, int lda);

void zher2_64(char uplo, long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy, doublecomplex *a, long lda);

PURPOSE

zher2 performs the hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of A is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of A is to be referenced.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of A is not referenced. On exit, the upper triangular part of the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of A is not referenced. On exit, the lower triangular part of the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDA (input)**

On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA >= max(1, n). Unchanged on exit.

NAME

zher2k - perform one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZHER2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER N, K, LDA, LDB, LDC
 DOUBLE PRECISION BETA

 SUBROUTINE ZHER2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDB, LDC
 DOUBLE PRECISION BETA
```

F95 INTERFACE

```
 SUBROUTINE HER2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: N, K, LDA, LDB, LDC
 REAL(8) :: BETA

 SUBROUTINE HER2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: N, K, LDA, LDB, LDC
```

```
REAL(8) :: BETA
```

C INTERFACE

#include <sunperf.h>

void zher2k(char uplo, char transa, int n, int k, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb, double beta, doublecomplex *c, int ldc);

void zher2k_64(char uplo, char transa, long n, long k, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb, double beta, doublecomplex *c, long ldc);

PURPOSE

zher2k K performs one of the Hermitian rank 2k operations C := alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C or C := alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C where alpha and beta are scalars with beta real, C is an n by n Hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*conjg(B') + conjg(alpha)*B*conjg(A') + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*conjg(A')*B + conjg(alpha)*conjg(B')*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'C' or 'c', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**

On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

  Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

zherfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZHERFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZHERFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HERFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HERFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zherfs(char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zherfs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zherfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CHETRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zherk - perform one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZHERK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX A(LDA,*), C(LDC,*)
 INTEGER N, K, LDA, LDC
 DOUBLE PRECISION ALPHA, BETA

 SUBROUTINE ZHERK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX A(LDA,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDC
 DOUBLE PRECISION ALPHA, BETA
```

F95 INTERFACE

```
 SUBROUTINE HERK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: N, K, LDA, LDC
 REAL(8) :: ALPHA, BETA

 SUBROUTINE HERK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: N, K, LDA, LDC
 REAL(8) :: ALPHA, BETA
```

C INTERFACE

#include <sunperf.h>

void zherk(char uplo, char transa, int n, int k, double alpha, doublecomplex *a, int lda, double beta, doublecomplex *c, int ldc);

void zherk_64(char uplo, char transa, long n, long k, double alpha, doublecomplex *a, long lda, double beta, doublecomplex *c, long ldc);

PURPOSE

zherk performs one of the Hermitian rank k operations C := alpha*A*conjg(A') + beta*C or C := alpha*conjg(A')*A + beta*C where alpha and beta are real scalars, C is an n by n Hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*conjg(A') + beta*C.

  TRANSA = 'C' or 'c' C : = alpha*conjg(A')*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'C' or 'c', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**

On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

  Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

zhesv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZHESV( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE ZHESV_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE HESV( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE HESV_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhesv(char uplo, int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zhesv_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zhesv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**H,  if UPLO = 'U', or

A = L * D * L**H,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CHETRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and

IPIVOT(k) = IPIVOT(k-1) < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and D(k-1:k,k-1:k) is a 2-by-2 diagonal block. If UPLO = 'L' and IPIVOT(k) = IPIVOT(k+1) < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and D(k:k+1,k:k+1) is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 1, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CHETRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

zhesvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZHESVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZHESVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HESVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HESVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
```

```
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhesvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zhesvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zhesvx uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
      A = U * D * U**H,  if UPLO = 'U', or

      A = L * D * L**H,  if UPLO = 'L',

   where U (or L) is a product of permutation and unit upper (lower)
   triangular matrices, and D is Hermitian and block diagonal with
   1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHETRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CHETRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CHETRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 2*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CHETRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zhetf2 - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZHETF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE ZHETF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE HETF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE HETF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zhetf2(char uplo, int n, doublecomplex *a, int lda, int *ipiv, int *info);

void zhetf2_64(char uplo, long n, doublecomplex *a, long lda, long *ipiv, long *info);

PURPOSE

zhetf2 computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the conjugate transpose of U, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by

```
  J. Lewis, Boeing Computer Services Company

  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

If UPLO = 'U', then A = U*D*U', where

```
   U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
            (   I    v    0   )   k-s

   U(k)  =  (   0    I    0   )   s

            (   0    0    I   )   n-k

               k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
   L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
            (   I    0     0   )  k-1

   L(k)  =  (   0    I     0   )  s

            (   0    v     I   )  n-k-s+1

               k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zhetrd - reduce a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
SUBROUTINE ZHETRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, LDA, LWORK, INFO
DOUBLE PRECISION D(*), E(*)

SUBROUTINE ZHETRD_64( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, LDA, LWORK, INFO
DOUBLE PRECISION D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE HETRD( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: D, E

SUBROUTINE HETRD_64( UPLO, [N], A, [LDA], D, E, TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, INFO
REAL(8), DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void zhetrd(char uplo, int n, doublecomplex *a, int lda, double *d, double *e, doublecomplex *tau, int *info);

void zhetrd_64(char uplo, long n, doublecomplex *a, long lda, double *d, double *e, doublecomplex *tau, long *info);

PURPOSE

zhetrd reduces a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1. For optimum performance LWORK >= N*NB, where NB is the optimal blocksize.

If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in A(1:i-1,i+1), and tau in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in A(i+2:n,i), and tau in TAU(i).

The contents of A on exit are illustrated by the following examples with n = 5:

if UPLO = 'U': if UPLO = 'L':

```
(  d   e   v2  v3  v4 )              (  d                  )
(      d   e   v3  v4 )              (  e   d              )
(          d   e   v4 )              (  v1  e   d          )
(              d   e  )              (  v1  v2  e   d      )
(                  d  )              (  v1  v2  v3  e   d  )
```

where d and e denote diagonal and off-diagonal elements of T, and vi denotes an element of the vector defining H(i).

NAME

zhetrf - compute the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZHETRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHETRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhetrf(char uplo, int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zhetrf_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zhetrf computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**H  or  A = L*D*L**H
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zhetri - compute the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
SUBROUTINE ZHETRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHETRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhetri(char uplo, int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zhetri_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zhetri computes the inverse of a complex Hermitian indefinite matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.

  On exit, if INFO = 0, the (Hermitian) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

zhetrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF

SYNOPSIS

```
SUBROUTINE ZHETRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHETRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HETRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HETRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhetrs(char uplo, int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zhetrs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zhetrs solves a system of linear equations A*X = B with a complex Hermitian matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHETRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHETRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHETRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zhgeqz - implement a single-shift version of the QZ method for finding the generalized eigenvalues `w(i)=ALPHA(i)/BETA(i)` of the equation det(A-w(i) B) = 0 If JOB='S', then the pair (A,B) is simultaneously reduced to Schur form (i.e., A and B are both upper triangular) by applying one unitary tranformation (usually called Q) on the left and another (usually called Z) on the right

---

SYNOPSIS

```
 SUBROUTINE ZHGEQZ( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZHGEQZ_64( JOB, COMPQ, COMPZ, N, ILO, IHI, A, LDA, B,
*      LDB, ALPHA, BETA, Q, LDQ, Z, LDZ, WORK, LWORK, RWORK, INFO)
 CHARACTER * 1 JOB, COMPQ, COMPZ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HGEQZ( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE HGEQZ_64( JOB, COMPQ, COMPZ, [N], ILO, IHI, A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], [WORK], [LWORK], [RWORK],
*       [INFO])
```

```
CHARACTER(LEN=1) :: JOB, COMPQ, COMPZ
COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
INTEGER(8) :: N, ILO, IHI, LDA, LDB, LDQ, LDZ, LWORK, INFO
REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zhgeqz(char job, char compq, char compz, int n, int ilo, int ihi, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *q, int ldq, doublecomplex *z, int ldz, int *info);

void zhgeqz_64(char job, char compq, char compz, long n, long ilo, long ihi, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *q, long ldq, doublecomplex *z, long ldz, long *info);

PURPOSE

zhgeqz implements a single-shift version of the QZ method for finding the generalized eigenvalues `w(i)=ALPHA(i)/BETA(i)` of the equation A are then ALPHA(1),...,ALPHA(N), and of B are BETA(1),...,BETA(N).

If JOB='S' and COMPQ and COMPZ are 'V' or 'I', then the unitary transformations used to reduce (A,B) are accumulated into the arrays Q and Z s.t.:

(in) `A(in)` Z(in)* = `Q(out)` `A(out)` Z(out)* (in) `B(in)` Z(in)* = `Q(out)` `B(out)` Z(out)*

Ref: C.B. Moler & G.W. Stewart, ``An Algorithm for Generalized Matrixigenvalue Problems'', SIAM J. Numer. Anal., 10(1973),p. 241--256.

ARGUMENTS

- **JOB (input)**

```
 = 'E': compute only ALPHA and BETA.  A and B will not
necessarily be put into generalized Schur form.
 = 'S': put A and B into generalized Schur form, as well
as computing ALPHA and BETA.
```

- **COMPQ (input)**

```
 = 'N': do not modify Q.

 = 'V': multiply the array Q on the right by the conjugate
transpose of the unitary tranformation that is
applied to the left side of A and B to reduce them
to Schur form.
 = 'I': like COMPQ ='V', except that Q will be initialized to
the identity first.
```

- **COMPZ (input)**

```
 = 'N': do not modify Z.

 = 'V': multiply the array Z on the right by the unitary
tranformation that is applied to the right side of
A and B to reduce them to Schur form.
 = 'I': like COMPZ ='V', except that Z will be initialized to
the identity first.
```

- **N (input)**
  The order of the matrices A, B, Q, and Z. N > = 0.
- **ILO (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  It is assumed that A is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **A (input/output)**
  On entry, the N-by-N upper Hessenberg matrix A. Elements below the subdiagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to upper triangular form. If JOB ='E', then on exit A will have been destroyed.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1, N).
- **B (input/output)**
  On entry, the N-by-N upper triangular matrix B. Elements below the diagonal must be zero. If JOB ='S', then on exit A and B will have been simultaneously reduced to upper triangular form. If JOB ='E', then on exit B will have been destroyed.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1, N).
- **ALPHA (output)**
  The diagonal elements of A when the pair (A,B) has been reduced to Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues.
- **BETA (output)**
  The diagonal elements of B when the pair (A,B) has been reduced to Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues. A and B are normalized so that `BETA(1),...,BETA(N)` are non-negative real numbers.
- **Q (input/output)**
  If COMPQ ='N', then Q will not be referenced. If COMPQ ='V' or 'I', then the conjugate transpose of the unitary transformations which are applied to A and B on the left will be applied to the array Q on the right.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1. If COMPQ ='V' or 'I', then LDQ > = N.
- **Z (input/output)**
  If COMPZ ='N', then Z will not be referenced. If COMPZ ='V' or 'I', then the unitary transformations which are applied to A and B on the right will be applied to the array Z on the right.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If COMPZ ='V' or 'I', then LDZ > = N.
- **WORK (workspace)**
  On exit, if INFO > = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **RWORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1,...,N: the QZ iteration did not converge.  (A,B) is not
in Schur form, but ALPHA(i) and BETA(i),
i =INFO+1,...,N should be correct.
 = N+1,...,2*N: the shift calculation failed.  (A,B) is not
in Schur form, but ALPHA(i) and BETA(i),
i =INFO-N+1,...,N should be correct.
 > 2*N:     various "impossible" errors.
```

FURTHER DETAILS

We assume that complex ABS works as long as its value is less than overflow.

NAME

zhpcon - estimate the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE ZHPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION ANORM, RCOND

SUBROUTINE ZHPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION ANORM, RCOND
```

F95 INTERFACE

```
SUBROUTINE HPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND

SUBROUTINE HPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void zhpcon(char uplo, int n, doublecomplex *a, int *ipivot, double anorm, double *rcond, int *info);

void zhpcon_64(char uplo, long n, doublecomplex *a, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zhpcon estimates the reciprocal of the condition number of a complex Hermitian packed matrix A using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zhpev - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage

---

SYNOPSIS

```
SUBROUTINE ZHPEV( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, WORK2, INFO)
CHARACTER * 1 JOBZ, UPLO
DOUBLE COMPLEX A(*), Z(LDZ,*), WORK(*)
INTEGER N, LDZ, INFO
DOUBLE PRECISION W(*), WORK2(*)

SUBROUTINE ZHPEV_64( JOBZ, UPLO, N, A, W, Z, LDZ, WORK, WORK2, INFO)
CHARACTER * 1 JOBZ, UPLO
DOUBLE COMPLEX A(*), Z(LDZ,*), WORK(*)
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE HPEV( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: W, WORK2

SUBROUTINE HPEV_64( JOBZ, UPLO, [N], A, W, Z, [LDZ], [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: JOBZ, UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhpev(char jobz, char uplo, int n, doublecomplex *a, double *w, doublecomplex *z, int ldz, int *info);

void zhpev_64(char jobz, char uplo, long n, doublecomplex *a, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhpev computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix in packed storage.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).

- **WORK (workspace)**
  `dimension(MAX(1,2*N-1))`

- **WORK2 (workspace)**
  `dimension(max(1,3*N-2))`

- **INFO (output)**

= 0: successful exit.

< 0: if INFO = -i, the i-th argument had an illegal value.

> 0: if INFO = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.

NAME

zhpevd - compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE ZHPEVD( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AP(*), Z(LDZ,*), WORK(*)
 INTEGER N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHPEVD_64( JOBZ, UPLO, N, AP, W, Z, LDZ, WORK, LWORK,
*      RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPEVD( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: AP, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HPEVD_64( JOBZ, UPLO, [N], AP, W, Z, [LDZ], WORK, [LWORK],
*       RWORK, [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: AP, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER(8) :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
```

```
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhpevd(char jobz, char uplo, int n, doublecomplex *ap, double *w, doublecomplex *z, int ldz, doublecomplex *work, int lwork, double *rwork, int lrwork, int *info);

void zhpevd_64(char jobz, char uplo, long n, doublecomplex *ap, double *w, doublecomplex *z, long ldz, doublecomplex *work, long lwork, double *rwork, long lrwork, long *info);

PURPOSE

zhpevd computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, AP is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.

- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the orthonormal eigenvectors of the matrix A, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (output)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of array WORK. If N < = 1, LWORK must be at least 1. If JOBZ = 'N' and N > 1, LWORK must be at least N. If JOBZ = 'V' and N > 1, LWORK must be at least 2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N < = 1, LRWORK must be at least 1. If JOBZ = 'N' and N > 1, LRWORK must be at least N. If JOBZ = 'V' and N > 1, LRWORK must be at least 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N < = 1, LIWORK must be at least 1. If JOBZ = 'V' and N > 1, LIWORK must be at least 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, the algorithm failed to converge; i
off-diagonal elements of an intermediate tridiagonal
form did not converge to zero.
```

NAME

zhpevx - compute selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage

---

SYNOPSIS

```
 SUBROUTINE ZHPEVX( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU, ABTOL,
*       NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(*), Z(LDZ,*), WORK(*)
 INTEGER N, IL, IU, NFOUND, LDZ, INFO
 INTEGER IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHPEVX_64( JOBZ, RANGE, UPLO, N, A, VL, VU, IL, IU,
*       ABTOL, NFOUND, W, Z, LDZ, WORK, WORK2, IWORK3, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX A(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, IL, IU, NFOUND, LDZ, INFO
 INTEGER*8 IWORK3(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABTOL
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPEVX( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU, ABTOL,
*       [NFOUND], W, Z, [LDZ], [WORK], [WORK2], [IWORK3], IFAIL, [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: A, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: N, IL, IU, NFOUND, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK3, IFAIL
 REAL(8) :: VL, VU, ABTOL
 REAL(8), DIMENSION(:) :: W, WORK2

 SUBROUTINE HPEVX_64( JOBZ, RANGE, UPLO, [N], A, VL, VU, IL, IU,
*       ABTOL, [NFOUND], W, Z, [LDZ], [WORK], [WORK2], [IWORK3], IFAIL,
*       [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, IL, IU, NFOUND, LDZ, INFO
INTEGER(8), DIMENSION(:) :: IWORK3, IFAIL
REAL(8) :: VL, VU, ABTOL
REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhpevx(char jobz, char range, char uplo, int n, doublecomplex *a, double vl, double vu, int il, int iu, double abtol, int *nfound, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zhpevx_64(char jobz, char range, char uplo, long n, doublecomplex *a, double vl, double vu, long il, long iu, double abtol, long *nfound, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zhpevx computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues/vectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **JOBZ (input)**

```
= 'N':  Compute eigenvalues only;

= 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
= 'A': all eigenvalues will be found;

= 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
= 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO

= 'L', A(i + (j-1)*(2*n-j)/2) = A(i,j) for j < =i < =n.

On exit, A is overwritten by values generated during the reduction to tridiagonal form. If UPLO = 'U', the diagonal and first superdiagonal of the tridiagonal matrix T overwrite the corresponding elements of A, and if UPLO = 'L', the diagonal and first subdiagonal of T overwrite the corresponding elements of A.

- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

  Eigenvalues will be computed most accurately when ABTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABTOL to 2*SLAMCH('S').

  See ``Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy,'' by Demmel and Kahan, LAPACK Working Note #3.
- **NFOUND (input)**
  The total number of eigenvalues found. 0 < = NFOUND < = N. If RANGE = 'A', NFOUND = N, and if RANGE = 'I', NFOUND = IU-IL+1.
- **W (output)**
  If INFO = 0, the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,NFOUND)` columns are supplied in the array Z; if RANGE = 'V', the exact value of NFOUND is not known in advance and an upper bound must be used.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(7*N)`
- **IWORK3 (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first NFOUND elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.

- **INFO (output)**

```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, then i eigenvectors failed to converge.
Their indices are stored in array IFAIL.
```

NAME

zhpgst - reduce a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage

---

SYNOPSIS

```
SUBROUTINE ZHPGST( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), BP(*)
INTEGER ITYPE, N, INFO

SUBROUTINE ZHPGST_64( ITYPE, UPLO, N, AP, BP, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), BP(*)
INTEGER*8 ITYPE, N, INFO
```

F95 INTERFACE

```
SUBROUTINE HPGST( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, BP
INTEGER :: ITYPE, N, INFO

SUBROUTINE HPGST_64( ITYPE, UPLO, N, AP, BP, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, BP
INTEGER(8) :: ITYPE, N, INFO
```

C INTERFACE

#include <sunperf.h>

void zhpgst(int itype, char uplo, int n, doublecomplex *ap, doublecomplex *bp, int *info);

void zhpgst_64(long itype, char uplo, long n, doublecomplex *ap, doublecomplex *bp, long *info);

PURPOSE

zhpgst reduces a complex Hermitian-definite generalized eigenproblem to standard form, using packed storage.

If ITYPE = 1, the problem is A*x = lambda*B*x,

and A is overwritten by `inv(U**H)*A*inv(U)` or `inv(L)*A*inv(L**H)`

If ITYPE = 2 or 3, the problem is A*B*x = lambda*x or

B*A*x = lambda*x, and A is overwritten by U*A*U**H or L**H*A*L.

B must have been previously factorized as U**H*U or L*L**H by CPPTRF.

ARGUMENTS

- **ITYPE (input)**

  ```
  = 1: compute inv(U**H)*A*inv(U) or inv(L)*A*inv(L**H);

  = 2 or 3: compute U*A*U**H or L**H*A*L.
  ```

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored and B is factored as
U**H*U;
  = 'L':  Lower triangle of A is stored and B is factored as
L*L**H.
  ```

- **N (input)**
  The order of the matrices A and B. N > = 0.

- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, if INFO = 0, the transformed matrix, stored in the same format as A.

- **BP (input)**
  The triangular factor from the Cholesky factorization of B, stored in the same format as A, as returned by CPPTRF.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zhpgv - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHPGV( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(*), B(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDZ, INFO
 DOUBLE PRECISION W(*), WORK2(*)

 SUBROUTINE ZHPGV_64( ITYPE, JOBZ, UPLO, N, A, B, W, Z, LDZ, WORK,
*      WORK2, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX A(*), B(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDZ, INFO
 DOUBLE PRECISION W(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGV( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: A, B, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK2

 SUBROUTINE HPGV_64( ITYPE, JOBZ, UPLO, [N], A, B, W, Z, [LDZ], [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: A, B, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER(8) :: ITYPE, N, LDZ, INFO
 REAL(8), DIMENSION(:) :: W, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhpgv(int itype, char jobz, char uplo, int n, doublecomplex *a, doublecomplex *b, double *w, doublecomplex *z, int ldz, int *info);

void zhpgv_64(long itype, char jobz, char uplo, long n, doublecomplex *a, doublecomplex *b, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhpgv computes all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;

  = 'L':  Lower triangles of A and B are stored.
  ```
- **N (input)**
  The order of the matrices A and B. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of A are destroyed.
- **B (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th

column of B is stored in the array B as follows: if UPLO = 'U', B(i + (j-1)*j/2) = B(i,j) for 1 <=i <=j; if UPLO = 'L', B(i + (j-1)*(2*n-j)/2) = B(i,j) for j <=i <=n.

On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.

- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  `dimension(MAX(1,2*N-1))`
- **WORK2 (workspace)**
  `dimension(MAX(1,3*N-2))`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEV returned an error code:

 < = N:  if INFO  = i, CHPEV failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not convergeto zero;
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

NAME

zhpgvd - compute all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHPGVD( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHPGVD_64( ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK,
*      LWORK, RWORK, LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, UPLO
 DOUBLE COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGVD( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ], [WORK],
*       [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
 COMPLEX(8), DIMENSION(:) :: AP, BP, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: W, RWORK

 SUBROUTINE HPGVD_64( ITYPE, JOBZ, UPLO, [N], AP, BP, W, Z, [LDZ],
*       [WORK], [LWORK], [RWORK], [LRWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, UPLO
```

```
COMPLEX(8), DIMENSION(:) :: AP, BP, WORK
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: ITYPE, N, LDZ, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhpgvd(int itype, char jobz, char uplo, int n, doublecomplex *ap, doublecomplex *bp, double *w, doublecomplex *z, int ldz, int *info);

void zhpgvd_64(long itype, char jobz, char uplo, long n, doublecomplex *ap, doublecomplex *bp, double *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhpgvd computes all the eigenvalues and, optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **UPLO (input)**

  ```
  = 'U':  Upper triangles of A and B are stored;
  ```

= 'L': Lower triangles of A and B are stored.

- **N (input)**
  The order of the matrices A and B. N >= 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 <=i <=j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j <=i <=n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.
- **W (output)**
  If INFO = 0, the eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, Z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I. If JOBZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of array WORK. If N <= 1, LWORK >= 1. If JOBZ = 'N' and N > 1, LWORK >= N. If JOBZ = 'V' and N > 1, LWORK >= 2*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (workspace)**
  On exit, if INFO = 0, `RWORK(1)` returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of array RWORK. If N <= 1, LRWORK >= 1. If JOBZ = 'N' and N > 1, LRWORK >= N. If JOBZ = 'V' and N > 1, LRWORK >= 1 + 5*N + 2*N**2.

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of array IWORK. If JOBZ = 'N' or N <= 1, LIWORK >= 1. If JOBZ = 'V' and N > 1, LIWORK >= 3 + 5*N.

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

= 0: successful exit

```
 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEVD returned an error code:

 < = N:  if INFO  = i, CHPEVD failed to converge;
i off-diagonal elements of an intermediate
tridiagonal form did not convergeto zero;
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

zhpgvx - compute selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x

---

SYNOPSIS

```
 SUBROUTINE ZHPGVX( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU, IL,
*      IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)

 SUBROUTINE ZHPGVX_64( ITYPE, JOBZ, RANGE, UPLO, N, AP, BP, VL, VU,
*      IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
 CHARACTER * 1 JOBZ, RANGE, UPLO
 DOUBLE COMPLEX AP(*), BP(*), Z(LDZ,*), WORK(*)
 INTEGER*8 ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER*8 IWORK(*), IFAIL(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION W(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HPGVX( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU, IL,
*       IU, ABSTOL, M, W, Z, [LDZ], [WORK], [RWORK], [IWORK], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: AP, BP, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

```
 SUBROUTINE HPGVX_64( ITYPE, JOBZ, RANGE, UPLO, [N], AP, BP, VL, VU,
*       IL, IU, ABSTOL, M, W, Z, [LDZ], [WORK], [RWORK], [IWORK], IFAIL,
*       [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE, UPLO
 COMPLEX(8), DIMENSION(:) :: AP, BP, WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER(8) :: ITYPE, N, IL, IU, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IWORK, IFAIL
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: W, RWORK
```

C INTERFACE

#include <sunperf.h>

void zhpgvx(int itype, char jobz, char range, char uplo, int n, doublecomplex *ap, doublecomplex *bp, double vl, double vu, int il, int iu, double abstol, int *m, double *w, doublecomplex *z, int ldz, int *ifail, int *info);

void zhpgvx_64(long itype, char jobz, char range, char uplo, long n, doublecomplex *ap, doublecomplex *bp, double vl, double vu, long il, long iu, double abstol, long *m, double *w, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zhpgvx computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x. Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

ARGUMENTS

- **ITYPE (input)**
  Specifies the problem type to be solved:

  ```
  = 1:  A*x  = (lambda)*B*x

  = 2:  A*B*x  = (lambda)*x

  = 3:  B*A*x  = (lambda)*x
  ```
- **JOBZ (input)**

  ```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
  ```
- **RANGE (input)**

  ```
  = 'A': all eigenvalues will be found;
  ```

```
 = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found;
 = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **UPLO (input)**

```
 = 'U':  Upper triangles of A and B are stored;

 = 'L':  Lower triangles of A and B are stored.
```

- **N (input)**
  The order of the matrices A and B. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.

  On exit, the contents of AP are destroyed.
- **BP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix B, packed columnwise in a linear array. The j-th column of B is stored in the array BP as follows: if UPLO = 'U', BP(i + (j-1)*j/2) = `B(i,j)` for 1 < =i < =j; if UPLO = 'L', BP(i + (j-1)*(2*n-j)/2) = `B(i,j)` for j < =i < =n.

  On exit, the triangular factor U or L from the Cholesky factorization B = U**H*U or B = L*L**H, in the same storage format as B.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

  ABSTOL + EPS * max(|a|,|b|) ,

  where EPS is the machine precision. If ABSTOL is less than or equal to zero, then EPS*|T| will be used in its place, where |T| is the 1-norm of the tridiagonal matrix obtained by reducing AP to tridiagonal form.

  Eigenvalues will be computed most accurately when ABSTOL is set to twice the underflow threshold 2*SLAMCH('S'), not zero. If this routine returns with INFO >0, indicating that some eigenvectors did not converge, try setting ABSTOL to 2*SLAMCH('S').
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  On normal exit, the first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'N', then Z is not referenced. If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of Z

holding the eigenvector associated with W(i). The eigenvectors are normalized as follows: if ITYPE = 1 or 2, Z**H*B*Z = I; if ITYPE = 3, Z**H*inv(B)*Z = I.

If an eigenvector fails to converge, then that column of Z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in IFAIL. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if JOBZ = 'V', LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(7*N)`
- **IWORK (workspace)**
  `dimension(5*N)`
- **IFAIL (output)**
  If JOBZ = 'V', then if INFO = 0, the first M elements of IFAIL are zero. If INFO > 0, then IFAIL contains the indices of the eigenvectors that failed to converge. If JOBZ = 'N', then IFAIL is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  CPPTRF or CHPEVX returned an error code:

 < = N:  if INFO  = i, CHPEVX failed to converge;
i eigenvectors failed to converge.  Their indices
are stored in array IFAIL.
 > N:   if INFO  = N + i, for 1  < = i  < = n, then the leading
minor of order i of B is not positive definite.
The factorization of B could not be completed and
no eigenvalues or eigenvectors were computed.
```

FURTHER DETAILS

Based on contributions by

```
   Mark Fahey, Department of Mathematics, Univ. of Kentucky, USA
```

NAME

zhpmv - perform the matrix-vector operation y := alpha*A*x + beta*y

---

SYNOPSIS

```
SUBROUTINE ZHPMV( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX A(*), X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE ZHPMV_64( UPLO, N, ALPHA, A, X, INCX, BETA, Y, INCY)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX A(*), X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE HPMV( UPLO, [N], ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: A, X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE HPMV_64( UPLO, [N], ALPHA, A, X, [INCX], BETA, Y, [INCY])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: A, X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zhpmv(char uplo, int n, doublecomplex alpha, doublecomplex *a, doublecomplex *x, int incx, doublecomplex beta, doublecomplex *y, int incy);

void zhpmv_64(char uplo, long n, doublecomplex alpha, doublecomplex *a, doublecomplex *x, long incx, doublecomplex beta, doublecomplex *y, long incy);

PURPOSE

zhpmv performs the matrix-vector operation y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. On exit, Y is overwritten by the updated vector y.

- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

zhpr - perform the hermitian rank 1 operation A := alpha*x*conjg(x') + A

SYNOPSIS

```
SUBROUTINE ZHPR( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
DOUBLE COMPLEX X(*), A(*)
INTEGER N, INCX
DOUBLE PRECISION ALPHA

SUBROUTINE ZHPR_64( UPLO, N, ALPHA, X, INCX, A)
CHARACTER * 1 UPLO
DOUBLE COMPLEX X(*), A(*)
INTEGER*8 N, INCX
DOUBLE PRECISION ALPHA
```

F95 INTERFACE

```
SUBROUTINE HPR( UPLO, [N], ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: X, A
INTEGER :: N, INCX
REAL(8) :: ALPHA

SUBROUTINE HPR_64( UPLO, [N], ALPHA, X, [INCX], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: X, A
INTEGER(8) :: N, INCX
REAL(8) :: ALPHA
```

C INTERFACE

#include <sunperf.h>

void zhpr(char uplo, int n, double alpha, doublecomplex *x, int incx, doublecomplex *a);

void zhpr_64(char uplo, long n, double alpha, doublecomplex *x, long incx, doublecomplex *a);

PURPOSE

zhpr performs the hermitian rank 1 operation A := alpha*x*conjg(x') + A where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

NAME

zhpr2 - perform the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A

SYNOPSIS

```
SUBROUTINE ZHPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(*)
INTEGER N, INCX, INCY

SUBROUTINE ZHPR2_64( UPLO, N, ALPHA, X, INCX, Y, INCY, A)
CHARACTER * 1 UPLO
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX X(*), Y(*), A(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE HPR2( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y, A
INTEGER :: N, INCX, INCY

SUBROUTINE HPR2_64( UPLO, [N], ALPHA, X, [INCX], Y, [INCY], A)
CHARACTER(LEN=1) :: UPLO
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: X, Y, A
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zhpr2(char uplo, int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy, doublecomplex *a);

void zhpr2_64(char uplo, long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy, doublecomplex *a);

PURPOSE

zhpr2 performs the Hermitian rank 2 operation A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array A as follows:

  UPLO = 'U' or 'u' The upper triangular part of A is supplied in A.

  UPLO = 'L' or 'l' The lower triangular part of A is supplied in A.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **X (input)**
  (1 + (n - 1)*abs(INCX)). Before entry, the incremented array X must contain the n element vector x. Unchanged on exit.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX < > 0. Unchanged on exit.
- **Y (input)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector y. Unchanged on exit.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.
- **A (input/output)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. On exit, the array A is overwritten by the upper triangular part of the updated matrix. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. On exit, the array A is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

NAME

zhprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZHPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZHPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPRFS( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE HPRFS_64( UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB], X,
*       [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhprfs(char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zhprfs_64(char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zhprfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CHPTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zhpsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZHPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPSV( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPSV_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhpsv(char uplo, int n, int nrhs, doublecomplex *a, int *ipivot, doublecomplex *b, int ldb, int *info);

void zhpsv_64(char uplo, long n, long nrhs, doublecomplex *a, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zhpsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**H,  if UPLO = 'U', or

A = L * D * L**H,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CHPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = conjg(aji))

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zhpsvx - use the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE ZHPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZHPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE HPSVX( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

```
 SUBROUTINE HPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zhpsvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zhpsvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zhpsvx uses the diagonal pivoting factorization A = U*D*U**H or A = L*D*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**H, if UPLO = 'U', or

```
    A = L * D * L**H,  if UPLO = 'L',

 where U (or L) is a product of permutation and unit upper (lower)
 triangular matrices and D is Hermitian and block diagonal with
 1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**H or A = L*D*L**H as computed by CHPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CHPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CHPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as

reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zhptrd - reduce a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation

---

SYNOPSIS

```
SUBROUTINE ZHPTRD( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), TAU(*)
INTEGER N, INFO
DOUBLE PRECISION D(*), E(*)

SUBROUTINE ZHPTRD_64( UPLO, N, AP, D, E, TAU, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), TAU(*)
INTEGER*8 N, INFO
DOUBLE PRECISION D(*), E(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRD( UPLO, [N], AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, TAU
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: D, E

SUBROUTINE HPTRD_64( UPLO, [N], AP, D, E, TAU, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, TAU
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: D, E
```

C INTERFACE

#include <sunperf.h>

void zhptrd(char uplo, int n, doublecomplex *ap, double *d, double *e, doublecomplex *tau, int *info);

void zhptrd_64(char uplo, long n, doublecomplex *ap, double *d, double *e, doublecomplex *tau, long *info);

PURPOSE

zhptrd reduces a complex Hermitian matrix A stored in packed form to real symmetric tridiagonal form T by a unitary similarity transformation: Q**H * A * Q = T.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **AP (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array AP as follows: if UPLO = 'U', AP(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', AP(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n. On exit, if UPLO = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors; if UPLO = 'L', the diagonal and first subdiagonal of A are over- written by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array TAU, represent the unitary matrix Q as a product of elementary reflectors. See Further Details.
- **D (output)**
  The diagonal elements of the tridiagonal matrix T: `D(i)` = A(i,i).
- **E (output)**
  The off-diagonal elements of the tridiagonal matrix T: `E(i)` = `A(i,i+1)` if UPLO = 'U', `E(i)` = `A(i+1,i)` if UPLO = 'L'.
- **TAU (output)**
  The scalar factors of the elementary reflectors (see Further Details).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

If UPLO = 'U', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(n-1) . . . H(2) H(1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(i+1:n)` = 0 and `v(i)` = 1; `v(1:i-1)` is stored on exit in AP, overwriting A(1:i-1,i+1), and tau is stored in TAU(i).

If UPLO = 'L', the matrix Q is represented as a product of elementary reflectors

```
Q  = H(1) H(2) . . . H(n-1).
```

Each `H(i)` has the form

```
H(i)  = I - tau * v * v'
```

where tau is a complex scalar, and v is a complex vector with `v(1:i)` = 0 and `v(i+1)` = 1; `v(i+2:n)` is stored on exit in AP, overwriting A(i+2:n,i), and tau is stored in TAU(i).

NAME

zhptrf - compute the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZHPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhptrf(char uplo, int n, doublecomplex *a, int *ipivot, int *info);

void zhptrf_64(char uplo, long n, doublecomplex *a, long *ipivot, long *info);

PURPOSE

zhptrf computes the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**H  or  A = L*D*L**H
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zhptri - compute the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE ZHPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRI( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRI_64( UPLO, [N], A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhptri(char uplo, int n, doublecomplex *a, int *ipivot, int *info);

void zhptri_64(char uplo, long n, doublecomplex *a, long *ipivot, long *info);

PURPOSE

zhptri computes the inverse of a complex Hermitian indefinite matrix A in packed storage using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (Hermitian) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.

- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.

- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

zhptrs - solve a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF

SYNOPSIS

```
SUBROUTINE ZHPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZHPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE HPTRS( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE HPTRS_64( UPLO, [N], [NRHS], A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zhptrs(char uplo, int n, int nrhs, doublecomplex *a, int *ipivot, doublecomplex *b, int ldb, int *info);

void zhptrs_64(char uplo, long n, long nrhs, doublecomplex *a, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zhptrs solves a system of linear equations A*X = B with a complex Hermitian matrix A stored in packed format using the factorization A = U*D*U**H or A = L*D*L**H computed by CHPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**H;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**H.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CHPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CHPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zhsein - use inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H

SYNOPSIS

```
 SUBROUTINE ZHSEIN( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, W, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 DOUBLE COMPLEX H(LDH,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER IFAILL(*), IFAILR(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZHSEIN_64( SIDE, EIGSRC, INITV, SELECT, N, H, LDH, W, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, IFAILL, IFAILR, INFO)
 CHARACTER * 1 SIDE, EIGSRC, INITV
 DOUBLE COMPLEX H(LDH,*), W(*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER*8 IFAILL(*), IFAILR(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE HSEIN( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], W, VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], IFAILL, IFAILR, [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: H, VL, VR
 INTEGER :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER, DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE HSEIN_64( SIDE, EIGSRC, INITV, SELECT, [N], H, [LDH], W,
```

```
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], IFAILL, IFAILR,
*       [INFO])
 CHARACTER(LEN=1) :: SIDE, EIGSRC, INITV
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: H, VL, VR
 INTEGER(8) :: N, LDH, LDVL, LDVR, MM, M, INFO
 INTEGER(8), DIMENSION(:) :: IFAILL, IFAILR
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void zhsein(char side, char eigsrc, char initv, logical *select, int n, doublecomplex *h, int ldh, doublecomplex *w, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int mm, int *m, int *ifaill, int *ifailr, int *info);

void zhsein_64(char side, char eigsrc, char initv, logical *select, long n, doublecomplex *h, long ldh, doublecomplex *w, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long mm, long *m, long *ifaill, long *ifailr, long *info);

PURPOSE

zhsein uses inverse iteration to find specified right and/or left eigenvectors of a complex upper Hessenberg matrix H.

The right eigenvector x and the left eigenvector y of the matrix H corresponding to an eigenvalue w are defined by:

```
H * x = w * x,     y**h * H = w * y**h
```

where y**h denotes the conjugate transpose of the vector y.

ARGUMENTS

- **SIDE (input)**

```
= 'R': compute right eigenvectors only;

= 'L': compute left eigenvectors only;

= 'B': compute both right and left eigenvectors.
```

- **EIGSRC (input)**
  Specifies the source of eigenvalues supplied in W:

```
 = 'Q': the eigenvalues were found using CHSEQR; thus, if
H has zero subdiagonal elements, and so is
block-triangular, then the j-th eigenvalue can be
assumed to be an eigenvalue of the block containing
the j-th row/column.  This property allows CHSEIN to
perform inverse iteration on just one diagonal block.
 = 'N': no assumptions are made on the correspondence
```

```
between eigenvalues and diagonal blocks.  In this
case, CHSEIN must always perform inverse iteration
using the whole matrix H.
```

- **INITV (input)**

```
 = 'N': no initial vectors are supplied;

 = 'U': user-supplied initial vectors are stored in the arrays
VL and/or VR.
```

- **SELECT (input)**
  Specifies the eigenvectors to be computed. To select the eigenvector corresponding to the eigenvalue W(j), `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrix H. N >= 0.
- **H (input)**
  The upper Hessenberg matrix H.
- **LDH (input)**
  The leading dimension of the array H. LDH >= max(1,N).
- **W (input/output)**
  On entry, the eigenvalues of H. On exit, the real parts of W may have been altered since close eigenvalues are perturbed slightly in searching for independent eigenvectors.
- **VL (input/output)**
  On entry, if INITV = 'U' and SIDE = 'L' or 'B', VL must contain starting vectors for the inverse iteration for the left eigenvectors; the starting vector for each eigenvector must be in the same column in which the eigenvector will be stored. On exit, if SIDE = 'L' or 'B', the left eigenvectors specified by SELECT will be stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if INITV = 'U' and SIDE = 'R' or 'B', VR must contain starting vectors for the inverse iteration for the right eigenvectors; the starting vector for each eigenvector must be in the same column in which the eigenvector will be stored. On exit, if SIDE = 'R' or 'B', the right eigenvectors specified by SELECT will be stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR required to store the eigenvectors (= the number of .TRUE. elements in SELECT).
- **WORK (workspace)**
  `dimension(N*N)`
- **RWORK (workspace)**
  `dimension(N)`
- **IFAILL (output)**
  If SIDE = 'L' or 'B', `IFAILL(i)` = j > 0 if the left eigenvector in the i-th column of VL (corresponding to the eigenvalue `w(j))` failed to converge; `IFAILL(i)` = 0 if the eigenvector converged satisfactorily. If SIDE = 'R', IFAILL is not referenced.
- **IFAILR (output)**
  If SIDE = 'R' or 'B', `IFAILR(i)` = j > 0 if the right eigenvector in the i-th column of VR (corresponding to the eigenvalue `w(j))` failed to converge; `IFAILR(i)` = 0 if the eigenvector converged satisfactorily. If SIDE = 'L', IFAILR is not referenced.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, i is the number of eigenvectors which
failed to converge; see IFAILL and IFAILR for further
details.
```

FURTHER DETAILS

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x|+|y|.

NAME

zhseqr - compute the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors

---

SYNOPSIS

```
 SUBROUTINE ZHSEQR( JOB, COMPZ, N, ILO, IHI, H, LDH, W, Z, LDZ, WORK,
*      LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 DOUBLE COMPLEX H(LDH,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER N, ILO, IHI, LDH, LDZ, LWORK, INFO

 SUBROUTINE ZHSEQR_64( JOB, COMPZ, N, ILO, IHI, H, LDH, W, Z, LDZ,
*      WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPZ
 DOUBLE COMPLEX H(LDH,*), W(*), Z(LDZ,*), WORK(*)
 INTEGER*8 N, ILO, IHI, LDH, LDZ, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE HSEQR( JOB, COMPZ, N, ILO, IHI, H, [LDH], W, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: H, Z
 INTEGER :: N, ILO, IHI, LDH, LDZ, LWORK, INFO

 SUBROUTINE HSEQR_64( JOB, COMPZ, N, ILO, IHI, H, [LDH], W, Z, [LDZ],
*       [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPZ
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: H, Z
 INTEGER(8) :: N, ILO, IHI, LDH, LDZ, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zhseqr(char job, char compz, int n, int ilo, int ihi, doublecomplex *h, int ldh, doublecomplex *w, doublecomplex *z, int ldz, int *info);

void zhseqr_64(char job, char compz, long n, long ilo, long ihi, doublecomplex *h, long ldh, doublecomplex *w, doublecomplex *z, long ldz, long *info);

PURPOSE

zhseqr computes the eigenvalues of a complex upper Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition H = Z T Z**H, where T is an upper triangular matrix (the Schur form), and Z is the unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input unitary matrix Q, so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the unitary matrix Q: A = Q*H*Q**H = (QZ)*T*(QZ)**H.

ARGUMENTS

- **JOB (input)**

```
  = 'E': compute eigenvalues only;

  = 'S': compute eigenvalues and the Schur form T.
```

- **COMPZ (input)**

```
  = 'N': no Schur vectors are computed;

  = 'I': Z is initialized to the unit matrix and the matrix Z
of Schur vectors of H is returned;
  = 'V': Z must contain an unitary matrix Q on entry, and
the product Q*Z is returned.
```

- **N (input)**
  The order of the matrix H. N > = 0.
- **ILO (input)**
  It is assumed that H is already upper triangular in rows and columns 1:ILO-1 and IHI+1:N. ILO and IHI are normally set by a previous call to CGEBAL, and then passed to CGEHRD when the matrix output by CGEBAL is reduced to Hessenberg form. Otherwise ILO and IHI should be set to 1 and N respectively. 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of ILO.
- **H (input/output)**
  On entry, the upper Hessenberg matrix H. On exit, if JOB = 'S', H contains the upper triangular matrix T from the Schur decomposition (the Schur form). If JOB = 'E', the contents of H are unspecified on exit.
- **LDH (input)**
  The leading dimension of the array H. LDH > = max(1,N).

- **W (output)**
  The computed eigenvalues. If JOB = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in H, with `W(i)` = H(i,i).

- **Z (input/output)**
  If COMPZ = 'N': Z is not referenced.

  If COMPZ = 'I': on entry, Z need not be set, and on exit, Z contains the unitary matrix Z of the Schur vectors of H. If COMPZ = 'V': on entry Z must contain an N-by-N matrix Q, which is assumed to be equal to the unit matrix except for the submatrix Z(ILO:IHI,ILO:IHI); on exit Z contains Q*Z. Normally Q is the unitary matrix generated by CUNGHR after the call to CGEHRD which formed the Hessenberg matrix H.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= `max(1,N)` if COMPZ = 'I' or 'V'; LDZ >= 1 otherwise.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.

- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.

- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, CHSEQR failed to compute all the
eigenvalues in a total of 30*(IHI-ILO+1) iterations;
elements 1:ilo-1 and i+1:n of W contain those
eigenvalues which have been successfully computed.
```

NAME

zlarz - applie a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right

---

SYNOPSIS

```
SUBROUTINE ZLARZ( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
DOUBLE COMPLEX TAU
DOUBLE COMPLEX V(*), C(LDC,*), WORK(*)
INTEGER M, N, L, INCV, LDC

SUBROUTINE ZLARZ_64( SIDE, M, N, L, V, INCV, TAU, C, LDC, WORK)
CHARACTER * 1 SIDE
DOUBLE COMPLEX TAU
DOUBLE COMPLEX V(*), C(LDC,*), WORK(*)
INTEGER*8 M, N, L, INCV, LDC
```

F95 INTERFACE

```
SUBROUTINE LARZ( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC], [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX(8) :: TAU
COMPLEX(8), DIMENSION(:) :: V, WORK
COMPLEX(8), DIMENSION(:,:) :: C
INTEGER :: M, N, L, INCV, LDC

SUBROUTINE LARZ_64( SIDE, [M], [N], L, V, [INCV], TAU, C, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX(8) :: TAU
COMPLEX(8), DIMENSION(:) :: V, WORK
COMPLEX(8), DIMENSION(:,:) :: C
INTEGER(8) :: M, N, L, INCV, LDC
```

C INTERFACE

#include <sunperf.h>

void zlarz(char side, int m, int n, int l, doublecomplex *v, int incv, doublecomplex tau, doublecomplex *c, int ldc);

void zlarz_64(char side, long m, long n, long l, doublecomplex *v, long incv, doublecomplex tau, doublecomplex *c, long ldc);

PURPOSE

zlarz applies a complex elementary reflector H to a complex M-by-N matrix C, from either the left or the right. H is represented in the form

```
H = I - tau * v * v'
```

where tau is a complex scalar and v is a complex vector.

If tau = 0, then H is taken to be the unit matrix.

To apply H' (the conjugate transpose of H), supply `conjg(tau)` instead tau.

H is a product of k elementary reflectors as returned by CTZRZF.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form  H * C

  = 'R': form  C * H
  ```
- **M (input)**
  The number of rows of the matrix C.
- **N (input)**
  The number of columns of the matrix C.
- **L (input)**
  The number of entries of the vector V containing the meaningful part of the Householder vectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  The vector v in the representation of H as returned by CTZRZF. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0.
- **TAU (input)**
  The value tau in the representation of H.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by the matrix H * C if SIDE = 'L', or C * H if SIDE = 'R'.
- **LDC (input)**

The leading dimension of the array C. LDC > = max(1,M).

- **WORK (workspace)**
  (N) if SIDE = 'L' or (M) if SIDE = 'R'

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

zlarzb - applie a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right

---

SYNOPSIS

```
 SUBROUTINE ZLARZB( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V, LDV,
*      T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 DOUBLE COMPLEX V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
 INTEGER M, N, K, L, LDV, LDT, LDC, LDWORK

 SUBROUTINE ZLARZB_64( SIDE, TRANS, DIRECT, STOREV, M, N, K, L, V,
*      LDV, T, LDT, C, LDC, WORK, LDWORK)
 CHARACTER * 1 SIDE, TRANS, DIRECT, STOREV
 DOUBLE COMPLEX V(LDV,*), T(LDT,*), C(LDC,*), WORK(LDWORK,*)
 INTEGER*8 M, N, K, L, LDV, LDT, LDC, LDWORK
```

F95 INTERFACE

```
 SUBROUTINE LARZB( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 COMPLEX(8), DIMENSION(:,:) :: V, T, C, WORK
 INTEGER :: M, N, K, L, LDV, LDT, LDC, LDWORK

 SUBROUTINE LARZB_64( SIDE, TRANS, DIRECT, STOREV, [M], [N], K, L, V,
*       [LDV], T, [LDT], C, [LDC], [WORK], [LDWORK])
 CHARACTER(LEN=1) :: SIDE, TRANS, DIRECT, STOREV
 COMPLEX(8), DIMENSION(:,:) :: V, T, C, WORK
 INTEGER(8) :: M, N, K, L, LDV, LDT, LDC, LDWORK
```

C INTERFACE

#include <sunperf.h>

void zlarzb(char side, char trans, char direct, char storev, int m, int n, int k, int l, doublecomplex *v, int ldv, doublecomplex *t, int ldt, doublecomplex *c, int ldc, int ldwork);

void zlarzb_64(char side, char trans, char direct, char storev, long m, long n, long k, long l, doublecomplex *v, long ldv, doublecomplex *t, long ldt, doublecomplex *c, long ldc, long ldwork);

PURPOSE

zlarzb applies a complex block reflector H or its transpose H**H to a complex distributed M-by-N C from the left or the right.

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **SIDE (input)**

  = 'L': apply H or H' from the Left

  = 'R': apply H or H' from the Right

- **TRANS (input)**

  = 'N': apply H (No transpose)

  = 'C': apply H' (Conjugate transpose)

- **DIRECT (input)**
  Indicates how H is formed from a product of elementary reflectors = 'F': H = `H(1) H(2) . . . H(k)` (Forward, not supported yet)

  = 'B': H = H(k) . . . H(2) H(1) (Backward)

- **STOREV (input)**
  Indicates how the vectors which define the elementary reflectors are stored:

  = 'C': Columnwise (not supported yet)

  = 'R': Rowwise

- **M (input)**
  The number of rows of the matrix C.

- **N (input)**
  The number of columns of the matrix C.

- **K (input)**
  The order of the matrix T (= the number of elementary reflectors whose product defines the block reflector).

- **L (input)**
  The number of columns of the matrix V containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **V (input)**
  If STOREV = 'C', NV = K; if STOREV = 'R', NV = L.
- **LDV (input)**
  The leading dimension of the array V. If STOREV = 'C', LDV > = L; if STOREV = 'R', LDV > = K.
- **T (input)**
  The triangular K-by-K matrix T in the representation of the block reflector.
- **LDT (input)**
  The leading dimension of the array T. LDT > = K.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by H*C or H'*C or C*H or C*H'.
- **LDC (input)**
  The leading dimension of the array C. LDC > = max(1,M).
- **WORK (workspace)**
  `dimension(MAX(M,N),K)`
- **LDWORK (input)**
  The leading dimension of the array WORK. If SIDE = 'L', LDWORK > = max(1,N); if SIDE = 'R', LDWORK > = max(1,M).

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

zlarzt - form the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors

---

SYNOPSIS

```
SUBROUTINE ZLARZT( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
DOUBLE COMPLEX V(LDV,*), TAU(*), T(LDT,*)
INTEGER N, K, LDV, LDT

SUBROUTINE ZLARZT_64( DIRECT, STOREV, N, K, V, LDV, TAU, T, LDT)
CHARACTER * 1 DIRECT, STOREV
DOUBLE COMPLEX V(LDV,*), TAU(*), T(LDT,*)
INTEGER*8 N, K, LDV, LDT
```

F95 INTERFACE

```
SUBROUTINE LARZT( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
COMPLEX(8), DIMENSION(:) :: TAU
COMPLEX(8), DIMENSION(:,:) :: V, T
INTEGER :: N, K, LDV, LDT

SUBROUTINE LARZT_64( DIRECT, STOREV, N, K, V, [LDV], TAU, T, [LDT])
CHARACTER(LEN=1) :: DIRECT, STOREV
COMPLEX(8), DIMENSION(:) :: TAU
COMPLEX(8), DIMENSION(:,:) :: V, T
INTEGER(8) :: N, K, LDV, LDT
```

C INTERFACE

#include <sunperf.h>

void zlarzt(char direct, char storev, int n, int k, doublecomplex *v, int ldv, doublecomplex *tau, doublecomplex *t, int ldt);

void zlarzt_64(char direct, char storev, long n, long k, doublecomplex *v, long ldv, doublecomplex *tau, doublecomplex *t, long ldt);

PURPOSE

zlarzt forms the triangular factor T of a complex block reflector H of order > n, which is defined as a product of k elementary reflectors.

If DIRECT = 'F', H = `H(1) H(2) . . . H(k)` and T is upper triangular;

If DIRECT = 'B', H = `H(k) . . . H(2) H(1)` and T is lower triangular.

If STOREV = 'C', the vector which defines the elementary reflector `H(i)` is stored in the i-th column of the array V, and

```
H  =  I - V * T * V'
```

If STOREV = 'R', the vector which defines the elementary reflector `H(i)` is stored in the i-th row of the array V, and

```
H  =  I - V' * T * V
```

Currently, only STOREV = 'R' and DIRECT = 'B' are supported.

ARGUMENTS

- **DIRECT (input)**
  Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

  ```
  = 'F': H  = H(1) H(2) . . . H(k) (Forward, not supported yet)

  = 'B': H  = H(k) . . . H(2) H(1) (Backward)
  ```
- **STOREV (input)**
  Specifies how the vectors which define the elementary reflectors are stored (see also Further Details):

  ```
  = 'R': rowwise
  ```
- **N (input)**
  The order of the block reflector H. N >= 0.
- **K (input)**
  The order of the triangular factor T (= the number of elementary reflectors). K >= 1.
- **V (input)**
  (LDV,K) if STOREV = 'C' (LDV,N) if STOREV = 'R' The matrix V. See further details.
- **LDV (input)**

The leading dimension of the array V. If STOREV = 'C', LDV > = max(1,N); if STOREV = 'R', LDV > = K.

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i).
- **T (output)**
  The k by k triangular factor T of the block reflector. If DIRECT = 'F', T is upper triangular; if DIRECT = 'B', T is lower triangular. The rest of the array is not used.
- **LDT (input)**
  The leading dimension of the array T. LDT > = K.

FURTHER DETAILS

Based on contributions by

```
  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The shape of the matrix V and the storage of the vectors which define the H(i) is best illustrated by the following example with n = 5 and k = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

DIRECT = 'F' and STOREV = 'C': DIRECT = 'F' and STOREV = 'R':

```
                                              _____V_____

     ( v1 v2 v3 )                            /           \
     ( v1 v2 v3 )                          ( v1 v1 v1 v1 v1 . . . . 1 )
V  = ( v1 v2 v3 )                           ( v2 v2 v2 v2 v2 . . . 1   )
     ( v1 v2 v3 )                          ( v3 v3 v3 v3 v3 . . 1     )
     ( v1 v2 v3 )

        .  .  .

        .  .  .

        1  .  .

           1  .

              1
```

DIRECT = 'B' and STOREV = 'C': DIRECT = 'B' and STOREV = 'R':

```
                                                  _____V_____
        1                                        /           \
        .  1                                  ( 1 . . . . v1 v1 v1 v1 v1 )
        .  .  1                               ( . 1 . . . v2 v2 v2 v2 v2 )
        .  .  .                               ( . . 1 . . v3 v3 v3 v3 v3 )
        .  .  .

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

```
V  = ( v1 v2 v3 )

     ( v1 v2 v3 )

     ( v1 v2 v3 )
```

NAME

zlatzm - routine is deprecated and has been replaced by routine CUNMRZ

SYNOPSIS

```
SUBROUTINE ZLATZM( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
DOUBLE COMPLEX TAU
DOUBLE COMPLEX V(*), C1(LDC,*), C2(LDC,*), WORK(*)
INTEGER M, N, INCV, LDC

SUBROUTINE ZLATZM_64( SIDE, M, N, V, INCV, TAU, C1, C2, LDC, WORK)
CHARACTER * 1 SIDE
DOUBLE COMPLEX TAU
DOUBLE COMPLEX V(*), C1(LDC,*), C2(LDC,*), WORK(*)
INTEGER*8 M, N, INCV, LDC
```

F95 INTERFACE

```
SUBROUTINE LATZM( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX(8) :: TAU
COMPLEX(8), DIMENSION(:) :: V, WORK
COMPLEX(8), DIMENSION(:,:) :: C1, C2
INTEGER :: M, N, INCV, LDC

SUBROUTINE LATZM_64( SIDE, [M], [N], V, [INCV], TAU, C1, C2, [LDC],
*       [WORK])
CHARACTER(LEN=1) :: SIDE
COMPLEX(8) :: TAU
COMPLEX(8), DIMENSION(:) :: V, WORK
COMPLEX(8), DIMENSION(:,:) :: C1, C2
INTEGER(8) :: M, N, INCV, LDC
```

C INTERFACE

#include <sunperf.h>

void zlatzm(char side, int m, int n, doublecomplex *v, int incv, doublecomplex tau, doublecomplex *c1, doublecomplex *c2, int ldc);

void zlatzm_64(char side, long m, long n, doublecomplex *v, long incv, doublecomplex tau, doublecomplex *c1, doublecomplex *c2, long ldc);

PURPOSE

zlatzm routine is deprecated and has been replaced by routine CUNMRZ.

CLATZM applies a Householder matrix generated by CTZRQF to a matrix.

Let P = I - tau*u*u', u = (1),

```
                                ( v )
```

where v is an (m-1) vector if SIDE = 'L', or a (n-1) vector if SIDE = 'R'.

If SIDE equals 'L', let

```
       C = [ C1 ] 1
           [ C2 ] m-1
             n
```

Then C is overwritten by P*C.

If SIDE equals 'R', let

```
       C = [ C1, C2 ] m
             1  n-1
```

Then C is overwritten by C*P.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': form P * C

  = 'R': form C * P
  ```

- **M (input)**
  The number of rows of the matrix C.
- **N (input)**
  The number of columns of the matrix C.
- **V (input)**
  (1 + (M-1)*abs(INCV)) if SIDE = 'L' (1 + (N-1)*abs(INCV)) if SIDE = 'R' The vector v in the representation of P. V is not used if TAU = 0.
- **INCV (input)**
  The increment between elements of v. INCV < > 0
- **TAU (input)**
  The value tau in the representation of P.
- **C1 (input/output)**
  (LDC,N) if SIDE = 'L' (M,1) if SIDE = 'R' On entry, the n-vector C1 if SIDE = 'L', or the m-vector C1 if SIDE = 'R'.

  On exit, the first row of P*C if SIDE = 'L', or the first column of C*P if SIDE = 'R'.
- **C2 (input/output)**
  (LDC, N) if SIDE = 'L' (LDC, N-1) if SIDE = 'R' On entry, the (m - 1) x n matrix C2 if SIDE = 'L', or the m x (n - 1) matrix C2 if SIDE = 'R'.

  On exit, rows 2:m of P*C if SIDE = 'L', or columns 2:m of C*P if SIDE = 'R'.
- **LDC (input)**
  The leading dimension of the arrays C1 and C2. LDC > = max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'

NAME

zpbcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

SYNOPSIS

```
 SUBROUTINE ZPBCON( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NDIAG, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZPBCON_64( UPLO, N, NDIAG, A, LDA, ANORM, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBCON( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE PBCON_64( UPLO, [N], NDIAG, A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zpbcon(char uplo, int n, int ndiag, doublecomplex *a, int lda, double anorm, double *rcond, int *info);

void zpbcon_64(char uplo, long n, long ndiag, doublecomplex *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

zpbcon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite band matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  `= 'U': Upper triangular factor stored in A;`

  `= 'L': Lower triangular factor stored in A.`
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian band matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**

dimension(N)

- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

zpbequ - compute row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
 SUBROUTINE ZPBEQU( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*)
 INTEGER N, NDIAG, LDA, INFO
 DOUBLE PRECISION SCOND, AMAX
 DOUBLE PRECISION SCALE(*)

 SUBROUTINE ZPBEQU_64( UPLO, N, NDIAG, A, LDA, SCALE, SCOND, AMAX,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*)
 INTEGER*8 N, NDIAG, LDA, INFO
 DOUBLE PRECISION SCOND, AMAX
 DOUBLE PRECISION SCALE(*)
```

F95 INTERFACE

```
 SUBROUTINE PBEQU( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL(8) :: SCOND, AMAX
 REAL(8), DIMENSION(:) :: SCALE

 SUBROUTINE PBEQU_64( UPLO, [N], NDIAG, A, [LDA], SCALE, SCOND, AMAX,
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, NDIAG, LDA, INFO
 REAL(8) :: SCOND, AMAX
 REAL(8), DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void zpbequ(char uplo, int n, int ndiag, doublecomplex *a, int lda, double *scale, double *scond, double *amax, int *info);

void zpbequ_64(char uplo, long n, long ndiag, doublecomplex *a, long lda, double *scale, double *scond, double *amax, long *info);

PURPOSE

zpbequ computes row and column scalings intended to equilibrate a Hermitian positive definite band matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular of A is stored;

= 'L':  Lower triangular of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit
```

< 0: if INFO = -i, the i-th argument had an illegal value.

> 0: if INFO = i, the i-th diagonal element is nonpositive.

NAME

zpbrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZPBRFS( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPBRFS_64( UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBRFS( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PBRFS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], AF, [LDAF],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zpbrfs(char uplo, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zpbrfs_64(char uplo, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zpbrfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and banded, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A as computed by CPBTRF, in the same storage format as A (see A).
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPBTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zpbstf - compute a split Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE ZPBSTF( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AB(LDAB,*)
INTEGER N, KD, LDAB, INFO

SUBROUTINE ZPBSTF_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AB(LDAB,*)
INTEGER*8 N, KD, LDAB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBSTF( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER :: N, KD, LDAB, INFO

SUBROUTINE PBSTF_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER(8) :: N, KD, LDAB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpbstf(char uplo, int n, int *kd, doublecomplex *ab, int ldab, int *info);

void zpbstf_64(char uplo, long n, long *kd, doublecomplex *ab, long ldab, long *info);

PURPOSE

zpbstf computes a split Cholesky factorization of a complex Hermitian positive definite band matrix A.

This routine is designed to be used in conjunction with CHBGST.

The factorization has the form A = S**H*S where S is a band matrix of the same bandwidth as A and the following structure:

```
S = ( U    )

    ( M  L )
```

where U is upper triangular of order m = (n+kd)/2, and L is lower triangular of order n-m.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangle of A is stored;

 = 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **KD (input/output)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. KD > = 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the factor S from the split Cholesky factorization A = S**H*S. See Further Details.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB > = KD+1.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the factorization could not be completed,
because the updated element a(i,i) was negative; the
matrix A is not positive definite.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 7, KD = 2:

S = (s11 s12 s13)

```
    (       s22  s23  s24                )
    (            s33  s34                )
    (                 s44                )
    (            s53  s54  s55           )
    (                 s64  s65  s66      )
    (                      s75  s76  s77 )
```

If UPLO = 'U', the array AB holds:

on entry: on exit:

```
 *    *   a13  a24  a35  a46  a57   *    *   s13  s24  s53' s64' s75'
 *   a12  a23  a34  a45  a56  a67   *   s12  s23  s34  s54' s65' s76'
a11  a22  a33  a44  a55  a66  a77  s11  s22  s33  s44  s55  s66  s77
```

If UPLO = 'L', the array AB holds:

on entry: on exit:

a11 a22 a33 a44 a55 a66 a77 s11 s22 s33 s44 s55 s66 s77 a21 a32 a43 a54 a65 a76 * s12' s23' s34' s54 s65 s76 * a31 a42 a53 a64 a64 * * s13' s24' s53 s64 s75 * *

Array elements marked * are not used by the routine; s12' denotes conjg(s12); the diagonal elements of S are real.

NAME

zpbsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZPBSV( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE ZPBSV_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBSV( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE PBSV_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpbsv(char uplo, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zpbsv_64(char uplo, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zpbsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite band matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H * U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular band matrix, and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i of A is not
positive definite, so the factorization could not be
completed, and the solution has not been computed.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
    *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
    *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
   a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
   a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
   a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
   a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

zpbsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZPBSVX( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPBSVX_64( FACT, UPLO, N, NDIAG, NRHS, A, LDA, AF, LDAF,
*      EQUED, SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2,
*      INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PBSVX( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

```
 SUBROUTINE PBSVX_64( FACT, UPLO, [N], NDIAG, [NRHS], A, [LDA], AF,
*       [LDAF], EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zpbsvx(char fact, char uplo, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, char equed, double *scale, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zpbsvx_64(char fact, char uplo, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, char equed, double *scale, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zpbsvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite band matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**H * U, if UPLO = 'U', or

```
   A = L * L**H,  if UPLO = 'L',

where U is an upper triangular band matrix, and L is a lower
triangular band matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.

- **NRHS (input)**
  The number of right-hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(NDIAG+1+i-j,j)` = `A(i,j)` for `max(1,j-NDIAG)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(N,j+NDIAG). See below for further details.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A (see A). If EQUED = 'Y', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = NDIAG+1.

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

```
 = 'Y':  Equilibration was done, i.e., A has been replaced by
diag(SCALE) * A * diag(SCALE).
EQUED is an input argument if FACT  = 'F'; otherwise, it is an
output argument.
```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
a11  a12  a13
     a22  a23  a24
          a33  a34  a35
               a44  a45  a46
                    a55  a56
(aij =conjg(aji))         a66
```

Band storage of the upper triangle of A:

```
 *    *   a13  a24  a35  a46
 *   a12  a23  a34  a45  a56
a11  a22  a33  a44  a55  a66
```

Similarly, if UPLO = 'L' the format of A is as follows:

```
a11  a22  a33  a44  a55  a66
a21  a32  a43  a54  a65   *
a31  a42  a53  a64   *    *
```

Array elements marked * are not used by the routine.

NAME

zpbtf2 - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE ZPBTF2( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AB(LDAB,*)
INTEGER N, KD, LDAB, INFO

SUBROUTINE ZPBTF2_64( UPLO, N, KD, AB, LDAB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AB(LDAB,*)
INTEGER*8 N, KD, LDAB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTF2( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER :: N, KD, LDAB, INFO

SUBROUTINE PBTF2_64( UPLO, [N], KD, AB, [LDAB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: AB
INTEGER(8) :: N, KD, LDAB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpbtf2(char uplo, int n, int kd, doublecomplex *ab, int ldab, int *info);

void zpbtf2_64(char uplo, long n, long kd, doublecomplex *ab, long ldab, long *info);

PURPOSE

zpbtf2 computes the Cholesky factorization of a complex Hermitian positive definite band matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix, U' is the conjugate transpose of U, and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **KD (input)**
  The number of super-diagonals of the matrix A if UPLO = 'U', or the number of sub-diagonals if UPLO = 'L'. KD >= 0.
- **AB (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first KD+1 rows of the array. The j-th column of A is stored in the j-th column of the array AB as follows: if UPLO = 'U', `AB(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `AB(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U'*U or A = L*L' of the band matrix A, in the same storage format as A.
- **LDAB (input)**
  The leading dimension of the array AB. LDAB >= KD+1.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, KD = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

NAME

zpbtrf - compute the Cholesky factorization of a complex Hermitian positive definite band matrix A

SYNOPSIS

```
SUBROUTINE ZPBTRF( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, NDIAG, LDA, INFO

SUBROUTINE ZPBTRF_64( UPLO, N, NDIAG, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, NDIAG, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTRF( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INFO

SUBROUTINE PBTRF_64( UPLO, [N], NDIAG, A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zpbtrf(char uplo, int n, int ndiag, doublecomplex *a, int lda, int *info);

void zpbtrf_64(char uplo, long n, long ndiag, doublecomplex *a, long lda, long *info);

PURPOSE

zpbtrf computes the Cholesky factorization of a complex Hermitian positive definite band matrix A.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j < =i < =min(n,j+kd).

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, in the same storage format as A.
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when N = 6, NDIAG = 2, and UPLO = 'U':

On entry: On exit:

```
 *    *   a13  a24  a35  a46      *    *   u13  u24  u35  u46
 *   a12  a23  a34  a45  a56      *   u12  u23  u34  u45  u56
a11  a22  a33  a44  a55  a66     u11  u22  u33  u44  u55  u66
```

Similarly, if UPLO = 'L' the format of A is as follows:

On entry: On exit:

```
a11  a22  a33  a44  a55  a66     l11  l22  l33  l44  l55  l66
a21  a32  a43  a54  a65   *      l21  l32  l43  l54  l65   *
a31  a42  a53  a64   *    *      l31  l42  l53  l64   *    *
```

Array elements marked * are not used by the routine.

Contributed by

Peter Mayes and Giuseppe Radicati, IBM ECSEC, Rome, March 23, 1989

NAME

zpbtrs - solve a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF

SYNOPSIS

```
SUBROUTINE ZPBTRS( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE ZPBTRS_64( UPLO, N, NDIAG, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PBTRS( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

SUBROUTINE PBTRS_64( UPLO, [N], NDIAG, [NRHS], A, [LDA], B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpbtrs(char uplo, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zpbtrs_64(char uplo, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zpbtrs solves a system of linear equations A*X = B with a Hermitian positive definite band matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPBTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangular factor stored in A;

= 'L':  Lower triangular factor stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals of the matrix A if UPLO = 'U', or the number of subdiagonals if UPLO = 'L'. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the band matrix A, stored in the first NDIAG+1 rows of the array. The j-th column of U or L is stored in the j-th column of the array A as follows: if UPLO ='U', `A(kd+1+i-j,j)` = `U(i,j)` for `max(1,j-kd)` < =i < =j; if UPLO ='L', `A(1+i-j,j)` = `L(i,j)` for j < =i < =min(n,j+kd).
- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zpocon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
 SUBROUTINE ZPOCON( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZPOCON_64( UPLO, N, A, LDA, ANORM, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 DOUBLE PRECISION ANORM, RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE POCON( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL(8) :: ANORM, RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE POCON_64( UPLO, [N], A, [LDA], ANORM, RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zpocon(char uplo, int n, doublecomplex *a, int lda, double anorm, double *rcond, int *info);

void zpocon_64(char uplo, long n, doublecomplex *a, long lda, double anorm, double *rcond, long *info);

PURPOSE

zpocon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zpoequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE ZPOEQU( N, A, LDA, SCALE, SCOND, AMAX, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION SCALE(*)

SUBROUTINE ZPOEQU_64( N, A, LDA, SCALE, SCOND, AMAX, INFO)
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE POEQU( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE

SUBROUTINE POEQU_64( [N], A, [LDA], SCALE, SCOND, AMAX, [INFO])
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void zpoequ(int n, doublecomplex *a, int lda, double *scale, double *scond, double *amax, int *info);

void zpoequ_64(long n, doublecomplex *a, long lda, double *scale, double *scond, double *amax, long *info);

PURPOSE

zpoequ computes row and column scalings intended to equilibrate a Hermitian positive definite matrix A and reduce its condition number (with respect to the two-norm). S contains the scale factors, `S(i)` = 1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)` = `S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The N-by-N Hermitian positive definite matrix whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the i-th diagonal element is nonpositive.
```

NAME

zporfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite,

---

SYNOPSIS

```
 SUBROUTINE ZPORFS( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPORFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PORFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PORFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], B,
*        [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zporfs(char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zporfs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zporfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPOTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zposv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO

SUBROUTINE ZPOSV_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE POSV( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO

SUBROUTINE POSV_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zposv(char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zposv_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zposv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H* U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i of A is not
  positive definite, so the factorization could not be
  completed, and the solution has not been computed.
  ```

NAME

zposvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZPOSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*      SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPOSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, EQUED,
*      SCALE, B, LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE POSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2

 SUBROUTINE POSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       EQUED, SCALE, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zposvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, char equed, double *scale, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zposvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, char equed, double *scale, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zposvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U**H* U, if UPLO = 'U', or

```
   A = L * L**H,  if UPLO = 'L',

where U is an upper triangular matrix and L is a lower triangular
matrix.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the Hermitian matrix A, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix diag(SCALE)*A*diag(SCALE).

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**

The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS righthand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zpotf2 - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

SYNOPSIS

```
SUBROUTINE ZPOTF2( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE ZPOTF2_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTF2( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTF2_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zpotf2(char uplo, int n, doublecomplex *a, int lda, int *info);

void zpotf2_64(char uplo, long n, doublecomplex *a, long lda, long *info);

PURPOSE

zpotf2 computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

The factorization has the form

```
A = U' * U ,  if UPLO = 'U', or

A = L  * L',  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading n by n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U'*U or A = L*L'.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value

  > 0: if INFO  = k, the leading minor of order k is not
positive definite, and the factorization could not be
completed.
  ```

NAME

zpotrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A

SYNOPSIS

```
SUBROUTINE ZPOTRF( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE ZPOTRF_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRF( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTRF_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zpotrf(char uplo, int n, doublecomplex *a, int lda, int *info);

void zpotrf_64(char uplo, long n, doublecomplex *a, long lda, long *info);

PURPOSE

zpotrf computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

This is the block version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the Hermitian matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the factorization could not be
completed.
  ```

NAME

zpotri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
SUBROUTINE ZPOTRI( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE ZPOTRI_64( UPLO, N, A, LDA, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRI( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE POTRI_64( UPLO, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zpotri(char uplo, int n, doublecomplex *a, int lda, int *info);

void zpotri_64(char uplo, long n, doublecomplex *a, long lda, long *info);

PURPOSE

zpotri computes the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF. On exit, the upper or lower triangle of the (Hermitian) inverse of A, overwriting the input factor U or L.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

zpotrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF

SYNOPSIS

```
SUBROUTINE ZPOTRS( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO

SUBROUTINE ZPOTRS_64( UPLO, N, NRHS, A, LDA, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE POTRS( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO

SUBROUTINE POTRS_64( UPLO, [N], [NRHS], A, [LDA], B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpotrs(char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void zpotrs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

zpotrs solves a system of linear equations A*X = B with a Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPOTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by CPOTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zppcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE ZPPCON( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION WORK2(*)

SUBROUTINE ZPPCON_64( UPLO, N, A, ANORM, RCOND, WORK, WORK2, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
SUBROUTINE PPCON( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2

SUBROUTINE PPCON_64( UPLO, N, A, ANORM, RCOND, [WORK], [WORK2],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void zppcon(char uplo, int n, doublecomplex *a, double anorm, double *rcond, int *info);

void zppcon_64(char uplo, long n, doublecomplex *a, double anorm, double *rcond, long *info);

PURPOSE

zppcon estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite packed matrix using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **ANORM (input)**
  The 1-norm (or infinity-norm) of the Hermitian matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zppequ - compute row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm)

SYNOPSIS

```
SUBROUTINE ZPPEQU( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER N, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION SCALE(*)

SUBROUTINE ZPPEQU_64( UPLO, N, A, SCALE, SCOND, AMAX, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
DOUBLE PRECISION SCOND, AMAX
DOUBLE PRECISION SCALE(*)
```

F95 INTERFACE

```
SUBROUTINE PPEQU( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE

SUBROUTINE PPEQU_64( UPLO, [N], A, SCALE, SCOND, AMAX, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
REAL(8) :: SCOND, AMAX
REAL(8), DIMENSION(:) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void zppequ(char uplo, int n, doublecomplex *a, double *scale, double *scond, double *amax, int *info);

void zppequ_64(char uplo, long n, doublecomplex *a, double *scale, double *scond, double *amax, long *info);

PURPOSE

zppequ computes row and column scalings intended to equilibrate a Hermitian positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). S contains the scale factors, S(i)=1/sqrt(A(i,i)), chosen so that the scaled matrix B with elements `B(i,j)=S(i)*A(i,j)*S(j)` has ones on the diagonal. This choice of S puts the condition number of B within a factor N of the smallest possible condition number over all possible diagonal scalings.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **SCALE (output)**
  If INFO = 0, SCALE contains the scale factors for A.
- **SCOND (output)**
  If INFO = 0, SCALE contains the ratio of the smallest `SCALE(i)` to the largest SCALE(i). If SCOND > = 0.1 and AMAX is neither too large nor too small, it is not worth scaling by SCALE.
- **AMAX (output)**
  Absolute value of largest matrix element. If AMAX is very close to overflow or very close to underflow, the matrix should be scaled.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

> 0:  if INFO  = i, the i-th diagonal element is nonpositive.
```

NAME

zpprfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZPPRFS( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR, BERR,
*      WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPPRFS_64( UPLO, N, NRHS, A, AF, B, LDB, X, LDX, FERR,
*      BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PPRFS( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX], FERR,
*       BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE PPRFS_64( UPLO, N, [NRHS], A, AF, B, [LDB], X, [LDX],
*       FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zpprfs(char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zpprfs_64(char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zpprfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, as computed by SPPTRF/CPPTRF, packed columnwise in a linear array in the same format as A (see A).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPPTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zppsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZPPSV( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE ZPPSV_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PPSV( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE PPSV_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zppsv(char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *b, int ldb, int *info);

void zppsv_64(char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *b, long ldb, long *info);

PURPOSE

zppsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

The Cholesky decomposition is used to factor A as

```
A = U**H* U,  if UPLO = 'U', or

A = L * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i of A is not
  positive definite, so the factorization could not be
  completed, and the solution has not been computed.
  ```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zppsvx - use the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZPPSVX( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPPSVX_64( FACT, UPLO, N, NRHS, A, AF, EQUED, SCALE, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO, EQUED
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION SCALE(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PPSVX( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2

 SUBROUTINE PPSVX_64( FACT, UPLO, [N], [NRHS], A, AF, EQUED, SCALE,
*       B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO, EQUED
```

```
COMPLEX(8), DIMENSION(:) :: A, AF, WORK
COMPLEX(8), DIMENSION(:,:) :: B, X
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: SCALE, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zppsvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, char equed, double *scale, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zppsvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, char equed, double *scale, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zppsvx uses the Cholesky factorization A = U**H*U or A = L*L**H to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N Hermitian positive definite matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'E', real scaling factors are computed to equilibrate the system:

```
   diag(S) * A * diag(S) * inv(diag(S)) * X = diag(S) * B
Whether or not the system will be equilibrated depends on the
scaling of the matrix A, but if equilibration is used, A is
overwritten by diag(S)*A*diag(S) and B by diag(S)*B.
```

2. If FACT = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if FACT = 'E') as A = U'* U , if UPLO = 'U', or

```
   A = L * L',  if UPLO = 'L',

where U is an upper triangular matrix, L is a lower triangular
matrix, and ' indicates conjugate transpose.
```

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by `diag(S)` so that it solves the original system before

```
equilibration.
```

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. = 'F': On entry, AF contains the factored form of A. If EQUED = 'Y', the matrix A has been equilibrated with scaling factors given by SCALE. A and AF will not be modified. = 'N': The matrix A will be copied to AF and factored.

  ```
   = 'E':  The matrix A will be equilibrated if necessary, then
  copied to AF and factored.
  ```

- **UPLO (input)**

  ```
   = 'U':  Upper triangle of A is stored;

   = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array, except if FACT = 'F' and EQUED = 'Y', then A must contain the equilibrated matrix diag(SCALE)*A*diag(SCALE). The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details. A is not modified if FACT = 'F' or 'N', or if FACT = 'E' and EQUED = 'N' on exit.

  On exit, if FACT = 'E' and EQUED = 'Y', A is overwritten by diag(SCALE)*A*diag(SCALE).

- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A. If EQUED .ne. 'N', then AF is the factored form of the equilibrated matrix A.

  If FACT = 'N', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the original matrix A.

  If FACT = 'E', then AF is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

- **EQUED (input)**
  Specifies the form of equilibration that was done. = 'N': No equilibration (always true if FACT = 'N').

  ```
   = 'Y':  Equilibration was done, i.e., A has been replaced by
  diag(SCALE) * A * diag(SCALE).
  EQUED is an input argument if FACT  = 'F'; otherwise, it is an
  output argument.
  ```

- **SCALE (input/output)**
  The scale factors for A; not accessed if EQUED = 'N'. SCALE is an input argument if FACT = 'F'; otherwise, SCALE is an output argument. If FACT = 'F' and EQUED = 'Y', each element of SCALE must be positive.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if EQUED = 'N', B is not modified; if EQUED = 'Y', B is overwritten by `diag(SCALE)` * B.

- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X to the original system of equations. Note that if EQUED = 'Y', A and B are modified on exit, and the solution to the equilibrated system is inv(diag(SCALE))*X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A after equilibration (if done). If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zpptrf - compute the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE ZPPTRF( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE ZPPTRF_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRF( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE PPTRF_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void zpptrf(char uplo, int n, doublecomplex *a, int *info);

void zpptrf_64(char uplo, long n, doublecomplex *a, long *info);

PURPOSE

zpptrf computes the Cholesky factorization of a complex Hermitian positive definite matrix A stored in packed format.

The factorization has the form

```
A = U**H * U,  if UPLO = 'U', or

A = L  * L**H,  if UPLO = 'L',
```

where U is an upper triangular matrix and L is lower triangular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. See below for further details.

  On exit, if INFO = 0, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, in the same storage format as A.

- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, the leading minor of order i is not
  positive definite, and the factorization could not be
  completed.
  ```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the Hermitian matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = conjg(aji))

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zpptri - compute the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE ZPPTRI( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE ZPPTRI_64( UPLO, N, A, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRI( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE PPTRI_64( UPLO, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void zpptri(char uplo, int n, doublecomplex *a, int *info);

void zpptri_64(char uplo, long n, doublecomplex *a, long *info);

PURPOSE

zpptri computes the inverse of a complex Hermitian positive definite matrix A using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

ARGUMENTS

- **UPLO (input)**

```
 = 'U':  Upper triangular factor is stored in A;

 = 'L':  Lower triangular factor is stored in A.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise as a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.

  On exit, the upper or lower triangle of the (Hermitian) inverse of A, overwriting the input factor U or L.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the (i,i) element of the factor U or L is
zero, and the inverse could not be computed.
```

NAME

zpptrs - solve a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF

SYNOPSIS

```
SUBROUTINE ZPPTRS( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE ZPPTRS_64( UPLO, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE PPTRS( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE PPTRS_64( UPLO, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void zpptrs(char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *b, int ldb, int *info);

void zpptrs_64(char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *b, long ldb, long *info);

PURPOSE

zpptrs solves a system of linear equations A*X = B with a Hermitian positive definite matrix A in packed storage using the Cholesky factorization A = U**H*U or A = L*L**H computed by CPPTRF.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The triangular factor U or L from the Cholesky factorization A = U**H*U or A = L*L**H, packed columnwise in a linear array. The j-th column of U or L is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `U(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `L(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zptcon - compute the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF

SYNOPSIS

```
SUBROUTINE ZPTCON( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
DOUBLE COMPLEX OFFD(*)
INTEGER N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION DIAG(*), WORK(*)

SUBROUTINE ZPTCON_64( N, DIAG, OFFD, ANORM, RCOND, WORK, INFO)
DOUBLE COMPLEX OFFD(*)
INTEGER*8 N, INFO
DOUBLE PRECISION ANORM, RCOND
DOUBLE PRECISION DIAG(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTCON( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: OFFD
INTEGER :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: DIAG, WORK

SUBROUTINE PTCON_64( [N], DIAG, OFFD, ANORM, RCOND, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: OFFD
INTEGER(8) :: N, INFO
REAL(8) :: ANORM, RCOND
REAL(8), DIMENSION(:) :: DIAG, WORK
```

C INTERFACE

#include <sunperf.h>

void zptcon(int n, double *diag, doublecomplex *offd, double anorm, double *rcond, int *info);

void zptcon_64(long n, double *diag, doublecomplex *offd, double anorm, double *rcond, long *info);

PURPOSE

zptcon computes the reciprocal of the condition number (in the 1-norm) of a complex Hermitian positive definite tridiagonal matrix using the factorization A = L*D*L**H or A = U**H*D*U computed by CPTTRF.

`Norm(inv(A))` is computed by a direct method, and the reciprocal of the condition number is computed as

```
RCOND = 1 / (ANORM * norm(inv(A))).
```

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization of A, as computed by CPTTRF.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization of A, as computed by CPTTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The method used is described in Nicholas J. Higham, ``Efficient Algorithms for Computing the Condition Number of a Tridiagonal Matrix'', SIAM J. Sci. Stat. Comput., Vol. 7, No. 1, January 1986.

NAME

zpteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor

---

SYNOPSIS

```
SUBROUTINE ZPTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
DOUBLE COMPLEX Z(LDZ,*)
INTEGER N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), WORK(*)

SUBROUTINE ZPTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
DOUBLE COMPLEX Z(LDZ,*)
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE PTEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK

SUBROUTINE PTEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void zpteqr(char compz, int n, double *d, double *e, doublecomplex *z, int ldz, int *info);

void zpteqr_64(char compz, long n, double *d, double *e, doublecomplex *z, long ldz, long *info);

PURPOSE

zpteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix by first factoring the matrix using SPTTRF and then calling CBDSQR to compute the singular values of the bidiagonal factor.

This routine computes the eigenvalues of the positive definite tridiagonal matrix to high relative accuracy. This means that if the eigenvalues range over many orders of magnitude in size, then the small eigenvalues and corresponding eigenvectors will be computed more accurately than, for example, with the standard QR method.

The eigenvectors of a full or band positive definite Hermitian matrix can also be found if CHETRD, CHPTRD, or CHBTRD has been used to reduce this matrix to tridiagonal form. (The reduction to tridiagonal form, however, may preclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix, if these eigenvalues range over many orders of magnitude.)

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvectors of original Hermitian
matrix also.  Array Z contains the unitary matrix
used to reduce the original matrix to tridiagonal
form.
 = 'I':  Compute eigenvectors of tridiagonal matrix also.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix. On normal exit, D contains the eigenvalues, in descending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', the unitary matrix used in the reduction to tridiagonal form. On exit, if COMPZ = 'V', the orthonormal eigenvectors of the original Hermitian matrix; if COMPZ = 'I', the orthonormal eigenvectors of the tridiagonal matrix. If INFO > 0 on exit, Z contains the eigenvectors associated with only the stored eigenvalues. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if COMPZ = 'V' or 'I', LDZ > = max(1,N).
- **WORK (workspace)**

dimension(4*N)

- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  if INFO  = i, and i is:
 < = N  the Cholesky factorization of the matrix could
not be performed because the i-th principal minor
was not positive definite.
 > N   the SVD algorithm failed to converge;
if INFO  = N+i, i off-diagonal elements of the
bidiagonal factor did not converge to zero.
```

NAME

zptrfs - improve the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZPTRFS( UPLO, N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX OFFD(*), OFFDF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPTRFS_64( UPLO, N, NRHS, DIAG, OFFD, DIAGF, OFFDF, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX OFFD(*), OFFDF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PTRFS( UPLO, [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: OFFD, OFFDF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2

 SUBROUTINE PTRFS_64( UPLO, [N], [NRHS], DIAG, OFFD, DIAGF, OFFDF, B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: OFFD, OFFDF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zptrfs(char uplo, int n, int nrhs, double *diag, doublecomplex *offd, double *diagf, doublecomplex *offdf, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zptrfs_64(char uplo, long n, long nrhs, double *diag, doublecomplex *offd, double *diagf, doublecomplex *offdf, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zptrfs improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian positive definite and tridiagonal, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:

  ```
  = 'U':  OFFD is the superdiagonal of A, and A  = U**H*DIAG*U;

  = 'L':  OFFD is the subdiagonal of A, and A  = L*DIAG*L**H.
  (The two forms are equivalent if A is real.)
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n real diagonal elements of the tridiagonal matrix A.
- **OFFD (input)**
  The (n-1) off-diagonal elements of the tridiagonal matrix A (see UPLO).
- **DIAGF (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization computed by CPTTRF.
- **OFFDF (input)**
  The (n-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by CPTTRF (see UPLO).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CPTTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).

- **FERR (output)**
  The forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

  = 0: successful exit

  < 0: if INFO = -i, the i-th argument had an illegal value

NAME

zptsv - compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

SYNOPSIS

```
SUBROUTINE ZPTSV( N, NRHS, DIAG, SUB, B, LDB, INFO)
DOUBLE COMPLEX SUB(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*)

SUBROUTINE ZPTSV_64( N, NRHS, DIAG, SUB, B, LDB, INFO)
DOUBLE COMPLEX SUB(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTSV( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:) :: SUB
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG

SUBROUTINE PTSV_64( [N], [NRHS], DIAG, SUB, B, [LDB], [INFO])
COMPLEX(8), DIMENSION(:) :: SUB
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void zptsv(int n, int nrhs, double *diag, doublecomplex *sub, doublecomplex *b, int ldb, int *info);

void zptsv_64(long n, long nrhs, double *diag, doublecomplex *sub, doublecomplex *b, long ldb, long *info);

PURPOSE

zptsv computes the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix, and X and B are N-by-NRHS matrices.

A is factored as A = L*D*L**H, and the factored form of A is then used to solve the system of equations.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the factorization A = L*DIAG*L**H.
- **SUB (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A. SUB can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U**H*DIAG*U factorization of A.
- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the leading minor of order i is not
positive definite, and the solution has not been
computed.  The factorization has not been completed
unless i  = N.
```

NAME

zptsvx - use the factorization A = L*D*L**H to compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE ZPTSVX( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT
 DOUBLE COMPLEX SUB(*), SUBF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZPTSVX_64( FACT, N, NRHS, DIAG, SUB, DIAGF, SUBF, B, LDB,
*      X, LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT
 DOUBLE COMPLEX SUB(*), SUBF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION DIAG(*), DIAGF(*), FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE PTSVX( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT
 COMPLEX(8), DIMENSION(:) :: SUB, SUBF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2

 SUBROUTINE PTSVX_64( FACT, [N], [NRHS], DIAG, SUB, DIAGF, SUBF, B,
*       [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT
 COMPLEX(8), DIMENSION(:) :: SUB, SUBF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: DIAG, DIAGF, FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zptsvx(char fact, int n, int nrhs, double *diag, doublecomplex *sub, double *diagf, doublecomplex *subf, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zptsvx_64(char fact, long n, long nrhs, double *diag, doublecomplex *sub, double *diagf, doublecomplex *subf, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zptsvx uses the factorization A = L*D*L**H to compute the solution to a complex system of linear equations A*X = B, where A is an N-by-N Hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the matrix A is factored as A = L*D*L**H, where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form

```
A = U**H*D*U.
```

2. If the leading i-by-i principal minor is not positive definite, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of the matrix A is supplied on entry. = 'F': On entry, DIAGF and SUBF contain the factored form of A. DIAG, SUB, DIAGF, and SUBF will not be modified. = 'N': The matrix A will be copied to DIAGF and SUBF and factored.
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the tridiagonal matrix A.

- **SUB (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix A.
- **DIAGF (input/output)**
  If FACT = 'F', then DIAGF is an input argument and on entry contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**H factorization of A. If FACT = 'N', then DIAGF is an output argument and on exit contains the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L**H factorization of A.
- **SUBF (input/output)**
  If FACT = 'F', then SUBF is an input argument and on entry contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A. If FACT = 'N', then SUBF is an output argument and on exit contains the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L**H factorization of A.
- **B (input)**
  On entry, the N-by-NRHS right hand side matrix B. Unchanged on exit.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **RCOND (output)**
  The reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j).
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  the leading minor of order i of A is
not positive definite, so the factorization
could not be completed, and the solution has not
been computed. RCOND  = 0 is returned.
 = N+1: U is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zpttrf - compute the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE ZPTTRF( N, DIAG, OFFD, INFO)
DOUBLE COMPLEX OFFD(*)
INTEGER N, INFO
DOUBLE PRECISION DIAG(*)

SUBROUTINE ZPTTRF_64( N, DIAG, OFFD, INFO)
DOUBLE COMPLEX OFFD(*)
INTEGER*8 N, INFO
DOUBLE PRECISION DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRF( [N], DIAG, OFFD, [INFO])
COMPLEX(8), DIMENSION(:) :: OFFD
INTEGER :: N, INFO
REAL(8), DIMENSION(:) :: DIAG

SUBROUTINE PTTRF_64( [N], DIAG, OFFD, [INFO])
COMPLEX(8), DIMENSION(:) :: OFFD
INTEGER(8) :: N, INFO
REAL(8), DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void zpttrf(int n, double *diag, doublecomplex *offd, int *info);

void zpttrf_64(long n, double *diag, doublecomplex *offd, long *info);

PURPOSE

zpttrf computes the L*D*L' factorization of a complex Hermitian positive definite tridiagonal matrix A. The factorization may also be regarded as having the form A = U'*D*U.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **DIAG (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix DIAG from the L*DIAG*L' factorization of A.
- **OFFD (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix A. On exit, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the L*DIAG*L' factorization of A. OFFD can also be regarded as the superdiagonal of the unit bidiagonal factor U from the U'*DIAG*U factorization of A.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, the leading minor of order k is not
positive definite; if k  < N, the factorization could not
be completed, while if k  = N, the factorization was
completed, but DIAG(N)  = 0.
```

NAME

zpttrs - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

SYNOPSIS

```
SUBROUTINE ZPTTRS( UPLO, N, NRHS, DIAG, OFFD, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX OFFD(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*)

SUBROUTINE ZPTTRS_64( UPLO, N, NRHS, DIAG, OFFD, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX OFFD(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
DOUBLE PRECISION DIAG(*)
```

F95 INTERFACE

```
SUBROUTINE PTTRS( UPLO, [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: OFFD
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG

SUBROUTINE PTTRS_64( UPLO, [N], [NRHS], DIAG, OFFD, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: OFFD
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
REAL(8), DIMENSION(:) :: DIAG
```

C INTERFACE

#include <sunperf.h>

void zpttrs(char uplo, int n, int nrhs, double *diag, doublecomplex *offd, doublecomplex *b, int ldb, int *info);

void zpttrs_64(char uplo, long n, long nrhs, double *diag, doublecomplex *offd, doublecomplex *b, long ldb, long *info);

PURPOSE

zpttrs solves a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF. D is a diagonal matrix specified in the vector D, U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **UPLO (input)**
  Specifies the form of the factorization and whether the vector OFFD is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L. = 'U': A = U'*DIAG*U, OFFD is the superdiagonal of U

  ```
  = 'L':  A  = L*DIAG*L', OFFD is the subdiagonal of L
  ```
- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **DIAG (input)**
  The n diagonal elements of the diagonal matrix DIAG from the factorization A = U'*DIAG*U or A = L*DIAG*L'.
- **OFFD (input/output)**
  If UPLO = 'U', the (n-1) superdiagonal elements of the unit bidiagonal factor U from the factorization A = U'*DIAG*U. If UPLO = 'L', the (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization A = L*DIAG*L'.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

zptts2 - solve a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF

SYNOPSIS

```
SUBROUTINE ZPTTS2( IUPLO, N, NRHS, D, E, B, LDB)
DOUBLE COMPLEX E(*), B(LDB,*)
INTEGER IUPLO, N, NRHS, LDB
DOUBLE PRECISION D(*)

SUBROUTINE ZPTTS2_64( IUPLO, N, NRHS, D, E, B, LDB)
DOUBLE COMPLEX E(*), B(LDB,*)
INTEGER*8 IUPLO, N, NRHS, LDB
DOUBLE PRECISION D(*)
```

F95 INTERFACE

```
SUBROUTINE ZPTTS2( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX(8), DIMENSION(:) :: E
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: IUPLO, N, NRHS, LDB
REAL(8), DIMENSION(:) :: D

SUBROUTINE ZPTTS2_64( IUPLO, N, NRHS, D, E, B, LDB)
COMPLEX(8), DIMENSION(:) :: E
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: IUPLO, N, NRHS, LDB
REAL(8), DIMENSION(:) :: D
```

C INTERFACE

#include <sunperf.h>

void zptts2(int iuplo, int n, int nrhs, double *d, doublecomplex *e, doublecomplex *b, int ldb);

void zptts2_64(long iuplo, long n, long nrhs, double *d, doublecomplex *e, doublecomplex *b, long ldb);

PURPOSE

zptts2 solves a tridiagonal system of the form A * X = B using the factorization A = U'*D*U or A = L*D*L' computed by CPTTRF. D is a diagonal matrix specified in the vector D, U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector E, and X and B are N by NRHS matrices.

ARGUMENTS

- **IUPLO (input)**
  Specifies the form of the factorization and whether the vector E is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L. = 1: A = U'*D*U, E is the superdiagonal of U

  ```
  = 0:  A  = L*D*L', E is the subdiagonal of L
  ```
- **N (input)**
  The order of the tridiagonal matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **D (input)**
  The n diagonal elements of the diagonal matrix D from the factorization A = U'*D*U or A = L*D*L'.
- **E (input)**
  If IUPLO = 1, the (n-1) superdiagonal elements of the unit bidiagonal factor U from the factorization A = U'*D*U.
  If IUPLO = 0, the (n-1) subdiagonal elements of the unit bidiagonal factor L from the factorization A = L*D*L'.
- **B (input/output)**
  On entry, the right hand side vectors B for the system of linear equations. On exit, the solution vectors, X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).

NAME

zrot - apply a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex

---

SYNOPSIS

```
SUBROUTINE ZROT( N, X, INCX, Y, INCY, C, S)
DOUBLE COMPLEX S
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY
DOUBLE PRECISION C

SUBROUTINE ZROT_64( N, X, INCX, Y, INCY, C, S)
DOUBLE COMPLEX S
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
DOUBLE PRECISION C
```

F95 INTERFACE

```
SUBROUTINE ROT( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX(8) :: S
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY
REAL(8) :: C

SUBROUTINE ROT_64( [N], X, [INCX], Y, [INCY], C, S)
COMPLEX(8) :: S
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
REAL(8) :: C
```

C INTERFACE

#include <sunperf.h>

void zrot(int n, doublecomplex *x, int incx, doublecomplex *y, int incy, double c, doublecomplex s);

void zrot_64(long n, doublecomplex *x, long incx, doublecomplex *y, long incy, double c, doublecomplex s);

PURPOSE

zrot applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors X and Y are complex.

ARGUMENTS

- **N (input)**
  The number of elements in the vectors X and Y.
- **X (output)**
  On input, the vector X. On output, X is overwritten with C*X + S*Y.
- **INCX (input)**
  The increment between successive values of Y. INCX < > 0.
- **Y (output)**
  On input, the vector Y. On output, Y is overwritten with -CONJG(S)*X + C*Y.
- **INCY (input)**
  The increment between successive values of Y. INCY < > 0.
- **C (input)**
- **S (input)**
  .Vb 5 \& C and S define a rotation \& [C S] \& [-conjg(S) C] \& \& where C*C + S*CONJG(S) = 1.0. .Ve

---

NAME

zrotg - Construct a Given's plane rotation

SYNOPSIS

```
SUBROUTINE ZROTG( A, B, C, S)
DOUBLE COMPLEX A, B, S
DOUBLE PRECISION C

SUBROUTINE ZROTG_64( A, B, C, S)
DOUBLE COMPLEX A, B, S
DOUBLE PRECISION C
```

F95 INTERFACE

```
SUBROUTINE ROTG( A, B, C, S)
COMPLEX(8) :: A, B, S
REAL(8) :: C

SUBROUTINE ROTG_64( A, B, C, S)
COMPLEX(8) :: A, B, S
REAL(8) :: C
```

C INTERFACE

#include <sunperf.h>

void zrotg(doublecomplex *a, doublecomplex b, double *c, doublecomplex *s);

void zrotg_64(doublecomplex *a, doublecomplex b, double *c, doublecomplex *s);

PURPOSE

zrotg Construct a Given's plane rotation that will annihilate an element of a vector.

ARGUMENTS

- **A (input/output)**
  On entry, A contains the entry in the first vector that corresponds to the element to be annihilated in the second vector. On exit, contains the nonzero element of the rotated vector.
- **B (input)**
  On entry, B contains the entry to be annihilated in the second vector. Unchanged on exit.
- **C (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.
- **S (output)**
  On exit, C and S are the elements of the rotation matrix that will be applied to annihilate B.

NAME

zscal - Compute y := alpha * y

SYNOPSIS

```
SUBROUTINE ZSCAL( N, ALPHA, Y, INCY)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX Y(*)
INTEGER N, INCY

SUBROUTINE ZSCAL_64( N, ALPHA, Y, INCY)
DOUBLE COMPLEX ALPHA
DOUBLE COMPLEX Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE SCAL( [N], ALPHA, Y, [INCY])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: Y
INTEGER :: N, INCY

SUBROUTINE SCAL_64( [N], ALPHA, Y, [INCY])
COMPLEX(8) :: ALPHA
COMPLEX(8), DIMENSION(:) :: Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void zscal(int n, doublecomplex alpha, doublecomplex *y, int incy);

void zscal_64(long n, doublecomplex alpha, doublecomplex *y, long incy);

PURPOSE

zscal Compute y := alpha * y where alpha is a scalar and y is an n-vector.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zsctr - Scatters elements from x into y.

SYNOPSIS

```
SUBROUTINE ZSCTR(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER NZ
INTEGER INDX(*)

SUBROUTINE ZSCTR_64(NZ, X, INDX, Y)

DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 NZ
INTEGER*8 INDX(*)
```

F95 INTERFACE SUBROUTINE SCTR([NZ], X, INDX, Y)

```
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: NZ
INTEGER, DIMENSION(:) :: INDX

SUBROUTINE SCTR_64([NZ], X, INDX, Y)

COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: NZ
INTEGER(8), DIMENSION(:) :: INDX
```

PURPOSE

ZSCTR - Scatters the components of a sparse vector x stored in compressed form into specified components of a vector y in full storage form.

```
do i = 1, n
  y(indx(i)) = x(i)
enddo
```

ARGUMENTS

NZ (input) - INTEGER

Number of elements in the compressed form. Unchanged on exit.

X (input)

Vector containing the values to be scattered from compressed form into full storage form. Unchanged on exit.

INDX (input) - INTEGER

Vector containing the indices of the compressed form. It is assumed that the elements in INDX are distinct and greater than zero. Unchanged on exit.

Y (output)

Vector whose elements specified by indx have been set to the corresponding entries of x. Only the elements corresponding to the indices in indx have been modified.

NAME

zspcon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE ZSPCON( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)
DOUBLE PRECISION ANORM, RCOND

SUBROUTINE ZSPCON_64( UPLO, N, A, IPIVOT, ANORM, RCOND, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
DOUBLE PRECISION ANORM, RCOND
```

F95 INTERFACE

```
SUBROUTINE SPCON( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND

SUBROUTINE SPCON_64( UPLO, [N], A, IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void zspcon(char uplo, int n, doublecomplex *a, int *ipivot, double anorm, double *rcond, int *info);

void zspcon_64(char uplo, long n, doublecomplex *a, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zspcon estimates the reciprocal of the condition number (in the 1-norm) of a complex symmetric packed matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zsprfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZSPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZSPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SPRFS( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SPRFS_64( UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X, [LDX],
*        FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
```

```
INTEGER(8) :: N, NRHS, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zsprfs(char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zsprfs_64(char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zsprfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite and packed, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input/output)**
  On entry, the solution matrix X, as computed by CSPTRS. On exit, the improved solution matrix X.

- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zspsv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
SUBROUTINE ZSPSV( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSPSV_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPSV( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPSV_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zspsv(char uplo, int n, int nrhs, doublecomplex *a, int *ipivot, doublecomplex *b, int ldb, int *info);

void zspsv_64(char uplo, long n, long nrhs, doublecomplex *a, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zspsv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be
computed.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
a11 a12 a13 a14

    a22 a23 a24

        a33 a34     (aij  = aji)

            a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zspsvx - use the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices

---

SYNOPSIS

```
 SUBROUTINE ZSPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZSPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, LDB, X,
*      LDX, RCOND, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(*), AF(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SPSVX( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB], X,
*       [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

```
 SUBROUTINE SPSVX_64( FACT, UPLO, N, NRHS, A, AF, IPIVOT, B, [LDB],
*       X, [LDX], RCOND, FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: A, AF, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zspsvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, doublecomplex *af, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zspsvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, doublecomplex *af, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zspsvx uses the diagonal pivoting factorization A = U*D*U**T or A = L*D*L**T to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix stored in packed format and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A as A = U * D * U**T, if UPLO = 'U', or

```
    A = L * D * L**T,  if UPLO = 'L',

 where U (or L) is a product of permutation and unit upper (lower)
 triangular matrices and D is symmetric and block diagonal with
 1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j <=i <=n. See below for further details.
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.

  If FACT = 'N', then AF is an output argument and on exit contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSPTRF, stored as a packed triangular matrix in the same storage format as A.
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CSPTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CSPTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the

largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.

- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

FURTHER DETAILS

The packed storage scheme is illustrated by the following example when N = 4, UPLO = 'U':

Two-dimensional storage of the symmetric matrix A:

```
   a11 a12 a13 a14

       a22 a23 a24

           a33 a34     (aij  = aji)

               a44
```

Packed storage of the upper triangle of A:

A = [a11, a12, a22, a13, a23, a33, a14, a24, a34, a44]

NAME

zsptrf - compute the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZSPTRF( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSPTRF_64( UPLO, N, A, IPIVOT, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRF( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRF_64( UPLO, [N], A, IPIVOT, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsptrf(char uplo, int n, doublecomplex *a, int *ipivot, int *info);

void zsptrf_64(char uplo, long n, doublecomplex *a, long *ipivot, long *info);

PURPOSE

zsptrf computes the factorization of a complex symmetric matrix A stored in packed format using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 <=i <=j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j <=i <=n.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L, stored as a packed triangular matrix overwriting A (see below for further details).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

```
 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

5-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zsptri - compute the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE ZSPTRI( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER N, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSPTRI_64( UPLO, N, A, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), WORK(*)
INTEGER*8 N, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRI( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRI_64( UPLO, N, A, IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A, WORK
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsptri(char uplo, int n, doublecomplex *a, int *ipivot, int *info);

void zsptri_64(char uplo, long n, doublecomplex *a, long *ipivot, long *info);

PURPOSE

zsptri computes the inverse of a complex symmetric indefinite matrix A in packed storage using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix, stored as a packed triangular matrix. The j-th column of `inv(A)` is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `inv(A)(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `inv(A)(i,j)` for j < =i < =n.

- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.

- **WORK (workspace)**
  `dimension(N)`

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

zsptrs - solve a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF

SYNOPSIS

```
SUBROUTINE ZSPTRS( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SPTRS( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SPTRS_64( UPLO, N, NRHS, A, IPIVOT, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsptrs(char uplo, int n, int nrhs, doublecomplex *a, int *ipivot, doublecomplex *b, int ldb, int *info);

void zsptrs_64(char uplo, long n, long nrhs, doublecomplex *a, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zsptrs solves a system of linear equations A*X = B with a complex symmetric matrix A stored in packed format using the factorization A = U*D*U**T or A = L*D*L**T computed by CSPTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSPTRF, stored as a packed triangular matrix.
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSPTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

zstedc - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method

---

SYNOPSIS

```
 SUBROUTINE ZSTEDC( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 COMPZ
 DOUBLE COMPLEX Z(LDZ,*), WORK(*)
 INTEGER N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION D(*), E(*), RWORK(*)

 SUBROUTINE ZSTEDC_64( COMPZ, N, D, E, Z, LDZ, WORK, LWORK, RWORK,
*      LRWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 COMPZ
 DOUBLE COMPLEX Z(LDZ,*), WORK(*)
 INTEGER*8 N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION D(*), E(*), RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEDC( COMPZ, [N], D, E, Z, [LDZ], [WORK], [LWORK],
*       RWORK, [LRWORK], IWORK, [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8), DIMENSION(:) :: D, E, RWORK

 SUBROUTINE STEDC_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [LWORK],
*       RWORK, [LRWORK], IWORK, [LIWORK], [INFO])
 CHARACTER(LEN=1) :: COMPZ
```

```
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, LWORK, LRWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8), DIMENSION(:) :: D, E, RWORK
```

C INTERFACE

#include <sunperf.h>

void zstedc(char compz, int n, double *d, double *e, doublecomplex *z, int ldz, double *rwork, int lrwork, int *iwork, int liwork, int *info);

void zstedc_64(char compz, long n, double *d, double *e, doublecomplex *z, long ldz, double *rwork, long lrwork, long *iwork, long liwork, long *info);

PURPOSE

zstedc computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band complex Hermitian matrix can also be found if CHETRD or CHPTRD or CHBTRD has been used to reduce this matrix to tridiagonal form.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none. See SLAED3 for details.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'I':  Compute eigenvectors of tridiagonal matrix also.

 = 'V':  Compute eigenvectors of original Hermitian matrix
also.  On entry, Z contains the unitary matrix used
to reduce the original matrix to tridiagonal form.
```

- **N (input)**
  The dimension of the symmetric tridiagonal matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the unitary matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original Hermitian matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z

is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1. If eigenvectors are desired, then LDZ > = max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If COMPZ = 'N' or 'I', or N < = 1, LWORK must be at least 1. If COMPZ = 'V' and N > 1, LWORK must be at least N*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **RWORK (output)**
  dimension (LRWORK) On exit, if INFO = 0, RWORK(1) returns the optimal LRWORK.
- **LRWORK (input)**
  The dimension of the array RWORK. If COMPZ = 'N' or N < = 1, LRWORK must be at least 1. If COMPZ = 'V' and N > 1, LRWORK must be at least 1 + 3*N + 2*N*lg N + 3*N**2 , where lg(N) = smallest integer k such that 2**k > = N. If COMPZ = 'I' and N > 1, LRWORK must be at least 1 + 4*N + 2*N**2 .

  If LRWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the RWORK array, returns this value as the first entry of the RWORK array, and no error message related to LRWORK is issued by XERBLA.
- **IWORK (output)**
  On exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. If COMPZ = 'N' or N < = 1, LIWORK must be at least 1. If COMPZ = 'V' or N > 1, LIWORK must be at least 6 + 6*N + 5*N*lg N. If COMPZ = 'I' or N > 1, LIWORK must be at least 3 + 5*N .

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.

 > 0:  The algorithm failed to compute an eigenvalue while
working on the submatrix lying in rows and columns
INFO/(N+1) through mod(INFO,N+1).
```

FURTHER DETAILS

Based on contributions by

```
   Jeff Rutter, Computer Science Division, University of California
   at Berkeley, USA
```

NAME

zstegr - Compute T-sigma_i = L_i D_i L_i^T, such that L_i D_i L_i^T is a relatively robust representation

---

SYNOPSIS

```
 SUBROUTINE ZSTEGR( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL, M,
*      W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 DOUBLE COMPLEX Z(LDZ,*)
 INTEGER N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)

 SUBROUTINE ZSTEGR_64( JOBZ, RANGE, N, D, E, VL, VU, IL, IU, ABSTOL,
*      M, W, Z, LDZ, ISUPPZ, WORK, LWORK, IWORK, LIWORK, INFO)
 CHARACTER * 1 JOBZ, RANGE
 DOUBLE COMPLEX Z(LDZ,*)
 INTEGER*8 N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER*8 ISUPPZ(*), IWORK(*)
 DOUBLE PRECISION VL, VU, ABSTOL
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEGR( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL, M,
*       W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
 CHARACTER(LEN=1) :: JOBZ, RANGE
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
 INTEGER, DIMENSION(:) :: ISUPPZ, IWORK
 REAL(8) :: VL, VU, ABSTOL
 REAL(8), DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEGR_64( JOBZ, RANGE, [N], D, E, VL, VU, IL, IU, ABSTOL,
*       M, W, Z, [LDZ], ISUPPZ, [WORK], [LWORK], [IWORK], [LIWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOBZ, RANGE
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, IL, IU, M, LDZ, LWORK, LIWORK, INFO
INTEGER(8), DIMENSION(:) :: ISUPPZ, IWORK
REAL(8) :: VL, VU, ABSTOL
REAL(8), DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void zstegr(char jobz, char range, int n, double *d, double *e, double vl, double vu, int il, int iu, double abstol, int *m, double *w, doublecomplex *z, int ldz, int *isuppz, int *info);

void zstegr_64(char jobz, char range, long n, double *d, double *e, double vl, double vu, long il, long iu, double abstol, long *m, double *w, doublecomplex *z, long ldz, long *isuppz, long *info);

PURPOSE

zstegr b) Compute the eigenvalues, lambda_j, of L_i D_i L_i^T to high relative accuracy by the dqds algorithm,

```
(c) If there is a cluster of close eigenvalues, "choose" sigma_i
    close to the cluster, and go to step (a),

(d) Given the approximate eigenvalue lambda_j of L_i D_i L_i^T,
    compute the corresponding eigenvector by forming a
    rank-revealing twisted factorization.
```

The desired accuracy of the output can be specified by the input parameter ABSTOL.

For more details, see ``A new `O(n^2)` algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", by Inderjit Dhillon, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

Note 1 : Currently CSTEGR is only set up to find ALL the n eigenvalues and eigenvectors of T in `O(n^2)` time

Note 2 : Currently the routine CSTEIN is called when an appropriate sigma_i cannot be chosen in step (c) above. CSTEIN invokes modified Gram-Schmidt when eigenvalues are close.

Note 3 : CSTEGR works only on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs. Normal execution of CSTEGR may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the ieee standard.

ARGUMENTS

- **JOBZ (input)**

```
  = 'N':  Compute eigenvalues only;

  = 'V':  Compute eigenvalues and eigenvectors.
```

- **RANGE (input)**

```
  = 'A': all eigenvalues will be found.

  = 'V': all eigenvalues in the half-open interval (VL,VU]
will be found.
  = 'I': the IL-th through IU-th eigenvalues will be found.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the n diagonal elements of the tridiagonal matrix T. On exit, D is overwritten.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix T in elements 1 to N-1 of E; `E(N)` need not be set. On exit, E is overwritten.
- **VL (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **VU (input)**
  If RANGE ='V', the lower and upper bounds of the interval to be searched for eigenvalues. VL < VU. Not referenced if RANGE = 'A' or 'I'.
- **IL (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **IU (input)**
  If RANGE ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 < = IL < = IU < = N, if N > 0; IL = 1 and IU = 0 if N = 0. Not referenced if RANGE = 'A' or 'V'.
- **ABSTOL (input)**
  The absolute error tolerance for the eigenvalues/eigenvectors. IF JOBZ = 'V', the eigenvalues and eigenvectors output have residual norms bounded by ABSTOL, and the dot products between different eigenvectors are bounded by ABSTOL. If ABSTOL is less than N*EPS*|T|, then N*EPS*|T| will be used in its place, where EPS is the machine precision and |T| is the 1-norm of the tridiagonal matrix. The eigenvalues are computed to an accuracy of EPS*|T| irrespective of ABSTOL. If high relative accuracy is important, set ABSTOL to DLAMCH('Safe minimum'). See Barlow and Demmel ``Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices'', LAPACK Working Note #7 for a discussion of which matrices define their eigenvalues to high relative accuracy.
- **M (output)**
  The total number of eigenvalues found. 0 < = M < = N. If RANGE = 'A', M = N, and if RANGE = 'I', M = IU-IL+1.
- **W (output)**
  The first M elements contain the selected eigenvalues in ascending order.
- **Z (output)**
  If JOBZ = 'V', then if INFO = 0, the first M columns of Z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with W(i). If JOBZ = 'N', then Z is not referenced. Note: the user must ensure that at least `max(1,M)` columns are supplied in the array Z; if RANGE = 'V', the exact value of M is not known in advance and an upper bound must be used.
- **LDZ (input)**

The leading dimension of the array Z. LDZ >= 1, and if JOBZ = 'V', LDZ >= max(1,N).

- **ISUPPZ (output)**
  The support of the eigenvectors in Z, i.e., the indices indicating the nonzero elements in Z. The i-th eigenvector is nonzero only in elements ISUPPZ(2*i-1) through ISUPPZ(2*i).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal (and minimal) LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= `max(1,18*N)`

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  On exit, if INFO = 0, `IWORK(1)` returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= `max(1,10*N)`

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = 1, internal error in SLARRE,
if INFO  = 2, internal error in CLARRV.
```

FURTHER DETAILS

Based on contributions by

```
Inderjit Dhillon, IBM Almaden, USA

Osni Marques, LBNL/NERSC, USA

Ken Stanley, Computer Science Division, University of California at Berkeley, USA
```

NAME

zstein - compute the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration

---

SYNOPSIS

```
 SUBROUTINE ZSTEIN( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 DOUBLE COMPLEX Z(LDZ,*)
 INTEGER N, M, LDZ, INFO
 INTEGER IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)

 SUBROUTINE ZSTEIN_64( N, D, E, M, W, IBLOCK, ISPLIT, Z, LDZ, WORK,
*      IWORK, IFAIL, INFO)
 DOUBLE COMPLEX Z(LDZ,*)
 INTEGER*8 N, M, LDZ, INFO
 INTEGER*8 IBLOCK(*), ISPLIT(*), IWORK(*), IFAIL(*)
 DOUBLE PRECISION D(*), E(*), W(*), WORK(*)
```

F95 INTERFACE

```
 SUBROUTINE STEIN( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ], [WORK],
*       [IWORK], IFAIL, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER :: N, M, LDZ, INFO
 INTEGER, DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL(8), DIMENSION(:) :: D, E, W, WORK

 SUBROUTINE STEIN_64( [N], D, E, [M], W, IBLOCK, ISPLIT, Z, [LDZ],
*       [WORK], [IWORK], IFAIL, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: Z
 INTEGER(8) :: N, M, LDZ, INFO
 INTEGER(8), DIMENSION(:) :: IBLOCK, ISPLIT, IWORK, IFAIL
 REAL(8), DIMENSION(:) :: D, E, W, WORK
```

C INTERFACE

#include <sunperf.h>

void zstein(int n, double *d, double *e, int m, double *w, int *iblock, int *isplit, doublecomplex *z, int ldz, int *ifail, int *info);

void zstein_64(long n, double *d, double *e, long m, double *w, long *iblock, long *isplit, doublecomplex *z, long ldz, long *ifail, long *info);

PURPOSE

zstein computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter MAXITS (currently set to 5).

Although the eigenvectors are real, they are stored in a complex array, which may be passed to CUNMTR or CUPMTR for back

transformation to the eigenvectors of a complex Hermitian matrix which was reduced to tridiagonal form.

ARGUMENTS

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input)**
  The n diagonal elements of the tridiagonal matrix T.
- **E (input)**
  The (n-1) subdiagonal elements of the tridiagonal matrix T, stored in elements 1 to N-1; `E(N)` need not be set.
- **M (input)**
  The number of eigenvectors to be found. 0 < = M < = N.
- **W (input)**
  The first M elements of W contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array W from SSTEBZ with ORDER = 'B' is expected here.)
- **IBLOCK (input)**
  The submatrix indices associated with the corresponding eigenvalues in W; `IBLOCK(i)` =1 if eigenvalue `W(i)` belongs to the first submatrix from the top, =2 if `W(i)` belongs to the second submatrix, etc. (The output array IBLOCK from SSTEBZ is expected here.)
- **ISPLIT (input)**
  The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to ISPLIT(1), the second of rows/columns ISPLIT(1)+1 through ISPLIT(2), etc. (The output array ISPLIT from SSTEBZ is expected here.)
- **Z (output)**
  The computed eigenvectors. The eigenvector associated with the eigenvalue `W(i)` is stored in the i-th column of Z. Any vector which fails to converge is set to its current iterate after MAXITS iterations. The imaginary parts of the

eigenvectors are set to zero.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(5*N)`
- **IWORK (workspace)**
  `dimension(N)`
- **IFAIL (output)**
  On normal exit, all elements of IFAIL are zero. If one or more eigenvectors fail to converge after MAXITS iterations, then their indices are stored in array IFAIL.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, then i eigenvectors failed to converge
in MAXITS iterations.  Their indices are stored in
array IFAIL.
```

NAME

zsteqr - compute all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method

---

SYNOPSIS

```
SUBROUTINE ZSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
DOUBLE COMPLEX Z(LDZ,*)
INTEGER N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), WORK(*)

SUBROUTINE ZSTEQR_64( COMPZ, N, D, E, Z, LDZ, WORK, INFO)
CHARACTER * 1 COMPZ
DOUBLE COMPLEX Z(LDZ,*)
INTEGER*8 N, LDZ, INFO
DOUBLE PRECISION D(*), E(*), WORK(*)
```

F95 INTERFACE

```
SUBROUTINE STEQR( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK

SUBROUTINE STEQR_64( COMPZ, [N], D, E, Z, [LDZ], [WORK], [INFO])
CHARACTER(LEN=1) :: COMPZ
COMPLEX(8), DIMENSION(:,:) :: Z
INTEGER(8) :: N, LDZ, INFO
REAL(8), DIMENSION(:) :: D, E, WORK
```

C INTERFACE

#include <sunperf.h>

void zsteqr(char compz, int n, double *d, double *e, doublecomplex *z, int ldz, int *info);

void zsteqr_64(char compz, long n, double *d, double *e, doublecomplex *z, long ldz, long *info);

PURPOSE

zsteqr computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. The eigenvectors of a full or band complex Hermitian matrix can also be found if CHETRD or CHPTRD or CHBTRD has been used to reduce this matrix to tridiagonal form.

ARGUMENTS

- **COMPZ (input)**

```
 = 'N':  Compute eigenvalues only.

 = 'V':  Compute eigenvalues and eigenvectors of the original
Hermitian matrix.  On entry, Z must contain the
unitary matrix used to reduce the original matrix
to tridiagonal form.
 = 'I':  Compute eigenvalues and eigenvectors of the
tridiagonal matrix.  Z is initialized to the identity
matrix.
```

- **N (input)**
  The order of the matrix. N > = 0.
- **D (input/output)**
  On entry, the diagonal elements of the tridiagonal matrix. On exit, if INFO = 0, the eigenvalues in ascending order.
- **E (input/output)**
  On entry, the (n-1) subdiagonal elements of the tridiagonal matrix. On exit, E has been destroyed.
- **Z (input/output)**
  On entry, if COMPZ = 'V', then Z contains the unitary matrix used in the reduction to tridiagonal form. On exit, if INFO = 0, then if COMPZ = 'V', Z contains the orthonormal eigenvectors of the original Hermitian matrix, and if COMPZ = 'I', Z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If COMPZ = 'N', then Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1, and if eigenvectors are desired, then LDZ > = max(1,N).
- **WORK (workspace)**
  `dimension(max(1,2*N-2))` If COMPZ = 'N', then WORK is not referenced.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value
```

> 0: the algorithm has failed to find all the eigenvalues in a total of 30*N iterations; if INFO = i, then i elements of E have not converged to zero; on exit, D and E contain the elements of a symmetric tridiagonal matrix which is unitarily similar to the original matrix.

NAME

zstsv - compute the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix

---

SYNOPSIS

```
SUBROUTINE ZSTSV( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)

SUBROUTINE ZSTSV_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STSV( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STSV_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zstsv(int n, int nrhs, doublecomplex *l, doublecomplex *d, doublecomplex *subl, doublecomplex *b, int ldb, int *ipiv, int *info);

void zstsv_64(long n, long nrhs, doublecomplex *l, doublecomplex *d, doublecomplex *subl, doublecomplex *b, long ldb, long *ipiv, long *info);

PURPOSE

zstsv computes the solution to a complex system of linear equations A * X = B where A is a Hermitian tridiagonal matrix.

ARGUMENTS

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides in B.
- **L (input/output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.
- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.
- **SUBL (output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On exit, part of the factorization of A.
- **B (input/output)**
  The columns of B contain the right hand sides.
- **LDB (input)**
  The leading dimension of B as specified in a type or DIMENSION statement.
- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.
- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

 > 0: if INFO = i, D(k,k) is exactly zero. The factorization has
been completed, but the block diagonal matrix D is exactly singular
and division by zero will occur if it is used to solve a system of
equations.

NAME

zsttrf - compute the factorization of a complex Hermitian tridiagonal matrix A

SYNOPSIS

```
SUBROUTINE ZSTTRF( N, L, D, SUBL, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*)
INTEGER N, INFO
INTEGER IPIV(*)

SUBROUTINE ZSTTRF_64( N, L, D, SUBL, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*)
INTEGER*8 N, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STTRF( [N], L, D, SUBL, IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
INTEGER :: N, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STTRF_64( [N], L, D, SUBL, IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
INTEGER(8) :: N, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zsttrf(int n, doublecomplex *l, doublecomplex *d, doublecomplex *subl, int *ipiv, int *info);

void zsttrf_64(long n, doublecomplex *l, doublecomplex *d, doublecomplex *subl, long *ipiv, long *info);

PURPOSE

zsttrf computes the L*D*L**H factorization of a complex Hermitian tridiagonal matrix A.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.

- **L (input/output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On entry, the n-1 subdiagonal elements of the tridiagonal matrix A. On exit, part of the factorization of A.

- **D (input/output)**

  ```
  REAL array, dimension (N)
  ```

  On entry, the n diagonal elements of the tridiagonal matrix A. On exit, the n diagonal elements of the diagonal matrix D from the factorization of A.

- **SUBL (output)**

  ```
  COMPLEX array, dimension (N)
  ```

  On exit, part of the factorization of A.

- **IPIV (output)**

  ```
  INTEGER array, dimension (N)
  ```

  On exit, the pivot indices of the factorization.

- **INFO (output)**

  ```
  INTEGER

  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value

  > 0:  if INFO  = i, D(k,k) is exactly zero.  The factorization has
  been completed, but the block diagonal matrix D is exactly singular
  and division by zero will occur if it is used to solve a system of
  equations.
  ```

NAME

zsttrs - computes the solution to a complex system of linear equations A * X = B

SYNOPSIS

```
SUBROUTINE ZSTTRS( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO
INTEGER IPIV(*)

SUBROUTINE ZSTTRS_64( N, NRHS, L, D, SUBL, B, LDB, IPIV, INFO)
DOUBLE COMPLEX L(*), D(*), SUBL(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE STTRS( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE STTRS_64( [N], [NRHS], L, D, SUBL, B, [LDB], IPIV, [INFO])
COMPLEX(8), DIMENSION(:) :: L, D, SUBL
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zsttrs(int n, int nrhs, doublecomplex *l, doublecomplex *d, doublecomplex *subl, doublecomplex *b, int ldb, int *ipiv, int *info);

void zsttrs_64(long n, long nrhs, doublecomplex *l, doublecomplex *d, doublecomplex *subl, doublecomplex *b, long ldb, long *ipiv, long *info);

PURPOSE

zsttrs computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric tridiagonal matrix and X and B are N-by-NRHS matrices.

ARGUMENTS

- **N (input)**

  ```
  INTEGER
  ```

  The order of the matrix A. N > = 0.
- **NRHS (input)**

  ```
  INTEGER
  ```

  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **L (input)**

  ```
  COMPLEX array, dimension (N-1)
  ```

  On entry, the subdiagonal elements of LL and DD.
- **D (input)**

  ```
  COMPLEX array, dimension (N)
  ```

  On entry, the diagonal elements of DD.
- **SUBL (input)**

  ```
  COMPLEX array, dimension (N-2)
  ```

  On entry, the second subdiagonal elements of LL.
- **B (input)**

  ```
  COMPLEX array, dimension (LDB, NRHS)
  ```

  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**

  ```
  INTEGER
  ```

  The leading dimension of the array B. LDB > = max(1, N)
- **IPIV (output)**

```
INTEGER array, dimension (N)
```

Details of the interchanges and block pivot. If IPIV(K) > 0, 1 by 1 pivot, and if IPIV(K) = K + 1 an interchange done; If IPIV(K) < 0, 2 by 2 pivot, no interchange required.

- **INFO (output)**

```
INTEGER

= 0:  successful exit

< 0:  if INFO  = -k, the k-th argument had an illegal value
```

NAME

zswap - Exchange vectors x and y.

SYNOPSIS

```
SUBROUTINE ZSWAP( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER N, INCX, INCY

SUBROUTINE ZSWAP_64( N, X, INCX, Y, INCY)
DOUBLE COMPLEX X(*), Y(*)
INTEGER*8 N, INCX, INCY
```

F95 INTERFACE

```
SUBROUTINE SWAP( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER :: N, INCX, INCY

SUBROUTINE SWAP_64( [N], X, [INCX], Y, [INCY])
COMPLEX(8), DIMENSION(:) :: X, Y
INTEGER(8) :: N, INCX, INCY
```

C INTERFACE

#include <sunperf.h>

void zswap(int n, doublecomplex *x, int incx, doublecomplex *y, int incy);

void zswap_64(long n, doublecomplex *x, long incx, doublecomplex *y, long incy);

PURPOSE

zswap Exchange x and y where x and y are n-vectors.

ARGUMENTS

- **N (input)**
  On entry, N specifies the number of elements in the vector. N must be at least one for the subroutine to have any visible effect. Unchanged on exit.
- **X (input/output)**
  (1 + (n - 1)*abs(INCX)). On entry, the incremented array X must contain the vector x. On exit, the y vector.
- **INCX (input)**
  On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). On entry, the incremented array Y must contain the vector y. On exit, the x vector.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

NAME

zsycon - estimate the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
 SUBROUTINE ZSYCON( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND

 SUBROUTINE ZSYCON_64( UPLO, N, A, LDA, IPIVOT, ANORM, RCOND, WORK,
*      INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION ANORM, RCOND
```

F95 INTERFACE

```
 SUBROUTINE SYCON( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND, [WORK],
*       [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: ANORM, RCOND

 SUBROUTINE SYCON_64( UPLO, [N], A, [LDA], IPIVOT, ANORM, RCOND,
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8) :: ANORM, RCOND
```

C INTERFACE

#include <sunperf.h>

void zsycon(char uplo, int n, doublecomplex *a, int lda, int *ipivot, double anorm, double *rcond, int *info);

void zsycon_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, double anorm, double *rcond, long *info);

PURPOSE

zsycon estimates the reciprocal of the condition number (in the 1-norm) of a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as RCOND = 1 / (ANORM * norm(inv(A))).

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **ANORM (input)**
  The 1-norm of the original matrix A.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(ANORM * AINVNM), where AINVNM is an estimate of the 1-norm of `inv(A)` computed in this routine.
- **WORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

  ```
  = 0:  successful exit
  ```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zsymm - perform one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZSYMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C,
*      LDC)
 CHARACTER * 1 SIDE, UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER M, N, LDA, LDB, LDC

 SUBROUTINE ZSYMM_64( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 SIDE, UPLO
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 M, N, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYMM( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: M, N, LDA, LDB, LDC

 SUBROUTINE SYMM_64( SIDE, UPLO, [M], [N], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: SIDE, UPLO
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: M, N, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void zsymm(char side, char uplo, int m, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex beta, doublecomplex *c, int ldc);

void zsymm_64(char side, char uplo, long m, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex beta, doublecomplex *c, long ldc);

PURPOSE

zsymm performs one of the matrix-matrix operations C := alpha*A*B + beta*C or C := alpha*B*A + beta*C where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether the symmetric matrix A appears on the left or right in the operation as follows:

  SIDE = 'L' or 'l' C : = alpha*A*B + beta*C,

  SIDE = 'R' or 'r' C : = alpha*B*A + beta*C,

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of the symmetric matrix is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of the symmetric matrix is to be referenced.

  Unchanged on exit.
- **M (input)**
  On entry, M specifies the number of rows of the matrix C. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of the matrix C. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  m when SIDE = 'L' or 'l' and is n otherwise.

  Before entry with SIDE = 'L' or 'l', the m by m part of the array A must contain the symmetric matrix, such that when UPLO = 'U' or 'u', the leading m by m upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading m by m lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

  Before entry with SIDE = 'R' or 'r', the n by n part of the array A must contain the symmetric matrix, such that when

UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced, and when UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1, m), otherwise LDA > = max(1, n). Unchanged on exit.
- **B (input)**
  Before entry, the leading m by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. LDB > = max(1, m). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. When BETA is supplied as zero then C need not be set on input. Unchanged on exit.
- **C (input/output)**
  Before entry, the leading m by n part of the array C must contain the matrix C, except when beta is zero, in which case C need not be set on entry. On exit, the array C is overwritten by the m by n updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC > = max(1, m). Unchanged on exit.

NAME

zsyr2k - perform one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZSYR2K( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB, BETA,
*      C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER N, K, LDA, LDB, LDC

 SUBROUTINE ZSYR2K_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, B, LDB,
*      BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDB, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYR2K( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B, [LDB],
*       BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: N, K, LDA, LDB, LDC

 SUBROUTINE SYR2K_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], B,
*       [LDB], BETA, C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: N, K, LDA, LDB, LDC
```

C INTERFACE

#include <sunperf.h>

void zsyr2k(char uplo, char transa, int n, int k, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex beta, doublecomplex *c, int ldc);

void zsyr2k_64(char uplo, char transa, long n, long k, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex beta, doublecomplex *c, long ldc);

PURPOSE

zsyr2k K performs one of the symmetric rank 2k operations C := alpha*A*B' + alpha*B*A' + beta*C or C := alpha*A'*B + alpha*B'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*B' + alpha*B*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*B + alpha*B'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrices A and B, and on entry with TRANSA = 'T' or 't', K specifies the number of rows of the matrices A and B. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.

- **B (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array B must contain the matrix B, otherwise the leading k by n part of the array B must contain the matrix B. Unchanged on exit.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDB must be at least max(1, n), otherwise LDB must be at least max(1, k). Unchanged on exit.
- **BETA (input)**
  On entry, BETA specifies the scalar beta. Unchanged on exit.
- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.
- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

zsyrfs - improve the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution

---

SYNOPSIS

```
 SUBROUTINE ZSYRFS( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZSYRFS_64( UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SYRFS( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF], IPIVOT,
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SYRFS_64( UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
```

```
INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zsyrfs(char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void zsyrfs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

zsyrfs improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **AF (input)**
  The factored form of the matrix A. AF contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**

The leading dimension of the array B. LDB >= max(1,N).

- **X (input/output)**
  On entry, the solution matrix X, as computed by CSYTRS. On exit, the improved solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zsyrk - perform one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C

SYNOPSIS

```
 SUBROUTINE ZSYRK( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C, LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), C(LDC,*)
 INTEGER N, K, LDA, LDC

 SUBROUTINE ZSYRK_64( UPLO, TRANSA, N, K, ALPHA, A, LDA, BETA, C,
*      LDC)
 CHARACTER * 1 UPLO, TRANSA
 DOUBLE COMPLEX ALPHA, BETA
 DOUBLE COMPLEX A(LDA,*), C(LDC,*)
 INTEGER*8 N, K, LDA, LDC
```

F95 INTERFACE

```
 SUBROUTINE SYRK( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA, C,
*       [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: N, K, LDA, LDC

 SUBROUTINE SYRK_64( UPLO, [TRANSA], [N], [K], ALPHA, A, [LDA], BETA,
*       C, [LDC])
 CHARACTER(LEN=1) :: UPLO, TRANSA
 COMPLEX(8) :: ALPHA, BETA
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: N, K, LDA, LDC
```

C INTERFACE

#include <sunperf.h>

void zsyrk(char uplo, char transa, int n, int k, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex beta, doublecomplex *c, int ldc);

void zsyrk_64(char uplo, char transa, long n, long k, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex beta, doublecomplex *c, long ldc);

PURPOSE

zsyrk performs one of the symmetric rank k operations C := alpha*A*A' + beta*C or C := alpha*A'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the upper or lower triangular part of the array C is to be referenced as follows:

  UPLO = 'U' or 'u' Only the upper triangular part of C is to be referenced.

  UPLO = 'L' or 'l' Only the lower triangular part of C is to be referenced.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' C : = alpha*A*A' + beta*C.

  TRANSA = 'T' or 't' C : = alpha*A'*A + beta*C.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix C. N must be at least zero. Unchanged on exit.
- **K (input)**
  On entry with TRANSA = 'N' or 'n', K specifies the number of columns of the matrix A, and on entry with TRANSA = 'T' or 't', K specifies the number of rows of the matrix A. K must be at least zero. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. Unchanged on exit.
- **A (input)**
  k when TRANSA = 'N' or 'n', and is n otherwise. Before entry with TRANSA = 'N' or 'n', the leading n by k part of the array A must contain the matrix A, otherwise the leading k by n part of the array A must contain the matrix A. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, n), otherwise LDA must be at least max(1, k). Unchanged on exit.
- **BETA (input)**

On entry, BETA specifies the scalar beta. Unchanged on exit.

- **C (input/output)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array C must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of the array C is overwritten by the upper triangular part of the updated matrix.

  Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array C must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of C is not referenced. On exit, the lower triangular part of the array C is overwritten by the lower triangular part of the updated matrix.

- **LDC (input)**
  On entry, LDC specifies the first dimension of C as declared in the calling (sub) program. LDC must be at least max(1, n). Unchanged on exit.

NAME

zsysv - compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZSYSV( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER IPIVOT(*)

 SUBROUTINE ZSYSV_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, WORK,
*      LDWORK, INFO)
 CHARACTER * 1 UPLO
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSV( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT

 SUBROUTINE SYSV_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [WORK], [LDWORK], [INFO])
 CHARACTER(LEN=1) :: UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsysv(char uplo, int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zsysv_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zsysv computes the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

The diagonal pivoting method is used to factor A as

```
A = U * D * U**T,  if UPLO = 'U', or

A = L * D * L**T,  if UPLO = 'L',
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. The factored form of A is then used to solve the system of equations A * X = B.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, if INFO = 0, the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIVOT (output)**
  Details of the interchanges and the block structure of D, as determined by CSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged, and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and

IPIVOT(k) = IPIVOT(k-1) < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and D(k-1:k,k-1:k) is a 2-by-2 diagonal block. If UPLO = 'L' and IPIVOT(k) = IPIVOT(k+1) < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and D(k:k+1,k:k+1) is a 2-by-2 diagonal block.

- **B (input/output)**
  On entry, the N-by-NRHS right hand side matrix B. On exit, if INFO = 0, the N-by-NRHS solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 1, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, so the solution could not be computed.
```

NAME

zsysvx - use the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZSYSVX( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT, B,
*      LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZSYSVX_64( FACT, UPLO, N, NRHS, A, LDA, AF, LDAF, IPIVOT,
*      B, LDB, X, LDX, RCOND, FERR, BERR, WORK, LDWORK, WORK2, INFO)
 CHARACTER * 1 FACT, UPLO
 DOUBLE COMPLEX A(LDA,*), AF(LDAF,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER*8 IPIVOT(*)
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE SYSVX( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER, DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE SYSVX_64( FACT, UPLO, [N], [NRHS], A, [LDA], AF, [LDAF],
*       IPIVOT, B, [LDB], X, [LDX], RCOND, FERR, BERR, [WORK], [LDWORK],
```

```
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: FACT, UPLO
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, AF, B, X
 INTEGER(8) :: N, NRHS, LDA, LDAF, LDB, LDX, LDWORK, INFO
 INTEGER(8), DIMENSION(:) :: IPIVOT
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void zsysvx(char fact, char uplo, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *af, int ldaf, int *ipivot, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *rcond, double *ferr, double *berr, int *info);

void zsysvx_64(char fact, char uplo, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *af, long ldaf, long *ipivot, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *rcond, double *ferr, double *berr, long *info);

PURPOSE

zsysvx uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations A * X = B, where A is an N-by-N symmetric matrix and X and B are N-by-NRHS matrices.

Error bounds on the solution and a condition estimate are also provided.

The following steps are performed:

1. If FACT = 'N', the diagonal pivoting method is used to factor A. The form of the factorization is

```
      A = U * D * U**T,  if UPLO = 'U', or

      A = L * D * L**T,  if UPLO = 'L',

   where U (or L) is a product of permutation and unit upper (lower)
   triangular matrices, and D is symmetric and block diagonal with
   1-by-1 and 2-by-2 diagonal blocks.
```

2. If some D(i,i)=0, so that D is exactly singular, then the routine returns with INFO = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, INFO = N+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

ARGUMENTS

- **FACT (input)**
  Specifies whether or not the factored form of A has been supplied on entry. = 'F': On entry, AF and IPIVOT contain the factored form of A. A, AF and IPIVOT will not be modified. = 'N': The matrix A will be copied to AF and factored.
- **UPLO (input)**

```
= 'U':  Upper triangle of A is stored;

= 'L':  Lower triangle of A is stored.
```

- **N (input)**
  The number of linear equations, i.e., the order of the matrix A. N >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >= 0.
- **A (input)**
  The symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **AF (input/output)**
  If FACT = 'F', then AF is an input argument and on entry contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T as computed by CSYTRF.

  If FACT = 'N', then AF is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization A = U*D*U**T or A = L*D*L**T.
- **LDAF (input)**
  The leading dimension of the array AF. LDAF >= max(1,N).
- **IPIVOT (input)**
  If FACT = 'F', then IPIVOT is an input argument and on entry contains details of the interchanges and the block structure of D, as determined by CSYTRF. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

  If FACT = 'N', then IPIVOT is an output argument and on exit contains details of the interchanges and the block structure of D, as determined by CSYTRF.
- **B (input)**
  The N-by-NRHS right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (output)**
  If INFO = 0 or INFO = N+1, the N-by-NRHS solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **RCOND (output)**
  The estimate of the reciprocal condition number of the matrix A. If RCOND is less than the machine precision (in particular, if RCOND = 0), the matrix is singular to working precision. This condition is indicated by a return code of INFO > 0.

- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LDWORK.
- **LDWORK (input)**
  The length of WORK. LDWORK >= 2*N, and for best performance LDWORK >= N*NB, where NB is the optimal blocksize for CSYTRF.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, and i is

 < = N:  D(i,i) is exactly zero.  The factorization
has been completed but the factor D is exactly
singular, so the solution and error bounds could
not be computed. RCOND  = 0 is returned.
 = N+1: D is nonsingular, but RCOND is less than machine
precision, meaning that the matrix is singular
to working precision.  Nevertheless, the
solution and error bounds are computed because
there are a number of situations where the
computed solution can be more accurate than the
value of RCOND would suggest.
```

NAME

zsytf2 - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZSYTF2( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO
INTEGER IPIV(*)

SUBROUTINE ZSYTF2_64( UPLO, N, A, LDA, IPIV, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIV(*)
```

F95 INTERFACE

```
SUBROUTINE SYTF2( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIV

SUBROUTINE SYTF2_64( UPLO, [N], A, [LDA], IPIV, [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIV
```

C INTERFACE

#include <sunperf.h>

void zsytf2(char uplo, int n, doublecomplex *a, int lda, int *ipiv, int *info);

void zsytf2_64(char uplo, long n, doublecomplex *a, long lda, long *ipiv, long *info);

PURPOSE

zsytf2 computes the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

```
A = U*D*U'  or  A = L*D*L'
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the transpose of U, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

  ```
  = 'U':  Upper triangular

  = 'L':  Lower triangular
  ```
- **N (input)**
  The order of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **IPIV (output)**
  Details of the interchanges and the block structure of D. If `IPIV(k)` > 0, then rows and columns k and `IPIV(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIV(k)` = `IPIV(k-1)` < 0, then rows and columns k-1 and -IPIV(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIV(k)` = `IPIV(k+1)` < 0, then rows and columns k+1 and -IPIV(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -k, the k-th argument had an illegal value

 > 0: if INFO  = k, D(k,k) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

1-96 - Based on modifications by J. Lewis, Boeing Computer Services Company

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    v    0   )   k-s

U(k)  =   (   0    I    0   )   s

          (   0    0    I   )   n-k

             k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIV(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
          (   I    0     0   )  k-1

L(k)  =   (   0    I     0   )  s

          (   0    v     I   )  n-k-s+1

             k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zsytrf - compute the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method

---

SYNOPSIS

```
SUBROUTINE ZSYTRF( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, LDWORK, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSYTRF_64( UPLO, N, A, LDA, IPIVOT, WORK, LDWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, LDWORK, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRF( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, LDWORK, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRF_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [LDWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LDWORK, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsytrf(char uplo, int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zsytrf_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zsytrf computes the factorization of a complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is

```
A = U*D*U**T  or  A = L*D*L**T
```

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  Upper triangle of A is stored;

  = 'L':  Lower triangle of A is stored.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.

- **A (input/output)**
  On entry, the symmetric matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

  On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L (see below for further details).

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).

- **IPIVOT (output)**
  Details of the interchanges and the block structure of D. If `IPIVOT(k)` > 0, then rows and columns k and `IPIVOT(k)` were interchanged and `D(k,k)` is a 1-by-1 diagonal block. If UPLO = 'U' and `IPIVOT(k)` = `IPIVOT(k-1)` < 0, then rows and columns k-1 and -IPIVOT(k) were interchanged and `D(k-1:k,k-1:k)` is a 2-by-2 diagonal block. If UPLO = 'L' and `IPIVOT(k)` = `IPIVOT(k+1)` < 0, then rows and columns k+1 and -IPIVOT(k) were interchanged and `D(k:k+1,k:k+1)` is a 2-by-2 diagonal block.

- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LDWORK.

- **LDWORK (input)**
  The length of WORK. LDWORK > =1. For best performance LDWORK > = N*NB, where NB is the block size returned by ILAENV.

  If LDWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LDWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, D(i,i) is exactly zero.  The factorization
has been completed, but the block diagonal matrix D is
exactly singular, and division by zero will occur if it
is used to solve a system of equations.
```

FURTHER DETAILS

If UPLO = 'U', then A = U*D*U', where

```
U  = P(n)*U(n)* ... *P(k)U(k)* ...,
```

i.e., U is a product of terms P(k)*U(k), where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `U(k)` is a unit upper triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    v    0   )   k-s

U(k)  =  (   0    I    0   )   s

         (   0    0    I   )   n-k

            k-s   s   n-k
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(1:k-1,k). If s = 2, the upper triangle of `D(k)` overwrites A(k-1,k-1), A(k-1,k), and A(k,k), and v overwrites A(1:k-2,k-1:k).

If UPLO = 'L', then A = L*D*L', where

```
L  = P(1)*L(1)* ... *P(k)*L(k)* ...,
```

i.e., L is a product of terms P(k)*L(k), where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(k). `P(k)` is a permutation matrix as defined by IPIVOT(k), and `L(k)` is a unit lower triangular matrix, such that if the diagonal block `D(k)` is of order s (s = 1 or 2), then

```
         (   I    0     0   )  k-1

L(k)  =  (   0    I     0   )  s
```

```
(   0    v     I   )  n-k-s+1

   k-1   s  n-k-s+1
```

If s = 1, `D(k)` overwrites A(k,k), and v overwrites A(k+1:n,k). If s = 2, the lower triangle of `D(k)` overwrites A(k,k), A(k+1,k), and A(k+1,k+1), and v overwrites A(k+2:n,k:k+1).

NAME

zsytri - compute the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
SUBROUTINE ZSYTRI( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER N, LDA, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSYTRI_64( UPLO, N, A, LDA, IPIVOT, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), WORK(*)
INTEGER*8 N, LDA, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRI( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRI_64( UPLO, [N], A, [LDA], IPIVOT, [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsytri(char uplo, int n, doublecomplex *a, int lda, int *ipivot, int *info);

void zsytri_64(char uplo, long n, doublecomplex *a, long lda, long *ipivot, long *info);

PURPOSE

zsytri computes the inverse of a complex symmetric indefinite matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **A (input/output)**
  On entry, the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.

  On exit, if INFO = 0, the (symmetric) inverse of the original matrix. If UPLO = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced; if UPLO = 'L' the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.

- **WORK (workspace)**
  `dimension(2*N)`

- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value

  > 0: if INFO  = i, D(i,i)  = 0; the matrix is singular and its
inverse could not be computed.
  ```

NAME

zsytrs - solve a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF

SYNOPSIS

```
SUBROUTINE ZSYTRS( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER N, NRHS, LDA, LDB, INFO
INTEGER IPIVOT(*)

SUBROUTINE ZSYTRS_64( UPLO, N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), B(LDB,*)
INTEGER*8 N, NRHS, LDA, LDB, INFO
INTEGER*8 IPIVOT(*)
```

F95 INTERFACE

```
SUBROUTINE SYTRS( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER :: N, NRHS, LDA, LDB, INFO
INTEGER, DIMENSION(:) :: IPIVOT

SUBROUTINE SYTRS_64( UPLO, [N], [NRHS], A, [LDA], IPIVOT, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:,:) :: A, B
INTEGER(8) :: N, NRHS, LDA, LDB, INFO
INTEGER(8), DIMENSION(:) :: IPIVOT
```

C INTERFACE

#include <sunperf.h>

void zsytrs(char uplo, int n, int nrhs, doublecomplex *a, int lda, int *ipivot, doublecomplex *b, int ldb, int *info);

void zsytrs_64(char uplo, long n, long nrhs, doublecomplex *a, long lda, long *ipivot, doublecomplex *b, long ldb, long *info);

PURPOSE

zsytrs solves a system of linear equations A*X = B with a complex symmetric matrix A using the factorization A = U*D*U**T or A = L*D*L**T computed by CSYTRF.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. = 'U': Upper triangular, form is A = U*D*U**T;

  ```
  = 'L':  Lower triangular, form is A  = L*D*L**T.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by CSYTRF.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **IPIVOT (input)**
  Details of the interchanges and the block structure of D as determined by CSYTRF.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

NAME

ztbcon - estimate the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE ZTBCON( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, NDIAG, LDA, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZTBCON_64( NORM, UPLO, DIAG, N, NDIAG, A, LDA, RCOND,
*      WORK, WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, NDIAG, LDA, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TBCON( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, NDIAG, LDA, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE TBCON_64( NORM, UPLO, DIAG, [N], NDIAG, A, [LDA], RCOND,
*       [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
```

```
INTEGER(8) :: N, NDIAG, LDA, INFO
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ztbcon(char norm, char uplo, char diag, int n, int ndiag, doublecomplex *a, int lda, double *rcond, int *info);

void ztbcon_64(char norm, char uplo, char diag, long n, long ndiag, doublecomplex *a, long lda, double *rcond, long *info);

PURPOSE

ztbcon estimates the reciprocal of the condition number of a triangular band matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <=j; if UPLO = 'L', `A(1+i-j,j)` = `A(i,j)` for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztbmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE ZTBMV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER N, NDIAG, LDA, INCY

SUBROUTINE ZTBMV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, NDIAG, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TBMV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INCY

SUBROUTINE TBMV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ztbmv(char uplo, char transa, char diag, int n, int ndiag, doublecomplex *a, int lda, doublecomplex *y, int incy);

void ztbmv_64(char uplo, char transa, char diag, long n, long ndiag, doublecomplex *a, long lda, doublecomplex *y, long incy);

PURPOSE

ztbmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the

matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztbrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE ZTBRFS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZTBRFS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TBRFS( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, X
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TBRFS_64( UPLO, [TRANSA], DIAG, [N], NDIAG, [NRHS], A,
*       [LDA], B, [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, X
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ztbrfs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void ztbrfs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

ztbrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular band coefficient matrix.

The solution matrix X must be computed by CTBTRS or some other means before entering this routine. CTBRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is

stored in the j-th column of the array A as follows: if UPLO = 'U', A(kd+1+i-j,j) = A(i,j) for max(1,j-kd) <=i <=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j <=i <=min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= NDIAG+1.
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector X(j) (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), FERR(j) is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector X(j) (i.e., the smallest relative change in any element of A or B that makes X(j) an exact solution).
- **WORK (workspace)**
  dimension(2*N)
- **WORK2 (workspace)**
  dimension(N)
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztbsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE ZTBSV( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER N, NDIAG, LDA, INCY

SUBROUTINE ZTBSV_64( UPLO, TRANSA, DIAG, N, NDIAG, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, NDIAG, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TBSV( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, NDIAG, LDA, INCY

SUBROUTINE TBSV_64( UPLO, [TRANSA], DIAG, [N], NDIAG, A, [LDA], Y,
*       [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, NDIAG, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ztbsv(char uplo, char transa, char diag, int n, int ndiag, doublecomplex *a, int lda, doublecomplex *y, int incy);

void ztbsv_64(char uplo, char transa, char diag, long n, long ndiag, doublecomplex *a, long lda, doublecomplex *y, long incy);

PURPOSE

ztbsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (ndiag + 1) diagonals.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **NDIAG (input)**
  On entry with UPLO = 'U' or 'u', NDIAG specifies the number of super-diagonals of the matrix A. On entry with UPLO = 'L' or 'l', NDIAG specifies the number of sub-diagonals of the matrix A. NDIAG > = 0. Unchanged on exit.
- **A (input)**

Before entry with UPLO = 'U' or 'u', the leading (ndiag + 1) by n part of the array A must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ndiag + 1) of the array, the first super-diagonal starting at position 2 in row ndiag, and so on. The top left ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = NDIAG + 1 - J
      DO 10, I = MAX( 1, J - NDIAG ), J
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Before entry with UPLO = 'L' or 'l', the leading (ndiag + 1) by n part of the array A must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right ndiag by ndiag triangle of the array A is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
    DO 20, J = 1, N
      M = 1 - J
      DO 10, I = J, MIN( N, J + NDIAG )
        A( M + I, J ) = matrix( I, J )
 10   CONTINUE
 20 CONTINUE
```

Note that when DIAG = 'U' or 'u' the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = (ndiag + 1). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztbtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZTBTRS( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NDIAG, NRHS, LDA, LDB, INFO

 SUBROUTINE ZTBTRS_64( UPLO, TRANSA, DIAG, N, NDIAG, NRHS, A, LDA, B,
*      LDB, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NDIAG, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
 SUBROUTINE TBTRS( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NDIAG, NRHS, LDA, LDB, INFO

 SUBROUTINE TBTRS_64( UPLO, TRANSA, DIAG, [N], NDIAG, [NRHS], A, [LDA],
*       B, [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NDIAG, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ztbtrs(char uplo, char transa, char diag, int n, int ndiag, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void ztbtrs_64(char uplo, char transa, char diag, long n, long ndiag, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

ztbtrs solves a triangular system of the form

where A is a triangular band matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N >= 0.
- **NDIAG (input)**
  The number of superdiagonals or subdiagonals of the triangular band matrix A. NDIAG >= 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
- **A (input)**
  The upper or lower triangular band matrix A, stored in the first kd+1 rows of A. The j-th column of A is stored in the j-th column of the array A as follows: if UPLO = 'U', `A(kd+1+i-j,j)` = `A(i,j)` for `max(1,j-kd)` <=i <

=j; if UPLO = 'L', A(1+i-j,j) = A(i,j) for j < =i < =min(n,j+kd). If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = NDIAG+1.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.
```

NAME

ztgevc - compute some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B)

SYNOPSIS

```
 SUBROUTINE ZTGEVC( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZTGEVC_64( SIDE, HOWMNY, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TGEVC( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE TGEVC_64( SIDE, HOWMNY, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
```

```
INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void ztgevc(char side, char howmny, logical *select, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int mm, int *m, int *info);

void ztgevc_64(char side, char howmny, logical *select, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

ztgevc computes some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A,B).

The right generalized eigenvector x and the left generalized eigenvector y of (A,B) corresponding to a generalized eigenvalue w are defined by:

```
(A - wB) * x = 0  and  y**H * (A - wB) = 0
```

where y**H denotes the conjugate tranpose of y.

If an eigenvalue w is determined by zero diagonal elements of both A and B, a unit vector is returned as the corresponding eigenvector.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of (A,B), or the products Z*X and/or Q*Y, where Z and Q are input unitary matrices. If (A,B) was obtained from the generalized Schur factorization of an original pair of matrices

```
(A0,B0) = (Q*A*Z**H,Q*B*Z**H),
```

then Z*X and Q*Y are the matrices of right or left eigenvectors of A.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'R': compute right eigenvectors only;

  = 'L': compute left eigenvectors only;

  = 'B': compute both right and left eigenvectors.
  ```

- **HOWMNY (input)**

```
 = 'A': compute all right and/or left eigenvectors;

 = 'B': compute all right and/or left eigenvectors, and
backtransform them using the input matrices supplied
in VR and/or VL;
 = 'S': compute selected right and/or left eigenvectors,
specified by the logical array SELECT.
```

- **SELECT (input)**
  If HOWMNY ='S', SELECT specifies the eigenvectors to be computed. If HOWMNY ='A' or 'B', SELECT is not referenced. To select the eigenvector corresponding to the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input)**
  The upper triangular matrix A.
- **LDA (input)**
  The leading dimension of array A. LDA >= max(1,N).
- **B (input)**
  The upper triangular matrix B. B must have real diagonal elements.
- **LDB (input)**
  The leading dimension of array B. LDB >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the unitary matrix Q of left Schur vectors returned by CHGEQZ). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of (A,B); if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the unitary matrix Z of right Schur vectors returned by CHGEQZ). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of (A,B); if HOWMNY = 'B', the matrix Z*X; if HOWMNY = 'S', the right eigenvectors of (A,B) specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected eigenvector occupies one column.
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(2*N)`
- **INFO (output)**

```
 = 0:  successful exit.

 < 0:  if INFO  = -i, the i-th argument had an illegal value.
```

NAME

ztgexc - reorder the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST

SYNOPSIS

```
 SUBROUTINE ZTGEXC( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z, LDZ,
*       IFST, ILST, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL WANTQ, WANTZ

 SUBROUTINE ZTGEXC_64( WANTQ, WANTZ, N, A, LDA, B, LDB, Q, LDQ, Z,
*       LDZ, IFST, ILST, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), Q(LDQ,*), Z(LDZ,*)
 INTEGER*8 N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL*8 WANTQ, WANTZ
```

F95 INTERFACE

```
 SUBROUTINE TGEXC( WANTQ, WANTZ, [N], A, [LDA], B, [LDB], Q, [LDQ],
*       Z, [LDZ], IFST, ILST, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL :: WANTQ, WANTZ

 SUBROUTINE TGEXC_64( WANTQ, WANTZ, [N], A, [LDA], B, [LDB], Q, [LDQ],
*       Z, [LDZ], IFST, ILST, [INFO])
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER(8) :: N, LDA, LDB, LDQ, LDZ, IFST, ILST, INFO
 LOGICAL(8) :: WANTQ, WANTZ
```

C INTERFACE

#include <sunperf.h>

void ztgexc(logical wantq, logical wantz, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *q, int ldq, doublecomplex *z, int ldz, int *ifst, int *ilst, int *info);

void ztgexc_64(logical wantq, logical wantz, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *q, long ldq, doublecomplex *z, long ldz, long *ifst, long *ilst, long *info);

PURPOSE

ztgexc reorders the generalized Schur decomposition of a complex matrix pair (A,B), using an unitary equivalence transformation (A, B) := Q * (A, B) * Z', so that the diagonal block of (A, B) with row index IFST is moved to row ILST.

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Optionally, the matrices Q and Z of generalized Schur vectors are updated.

```
Q(in) * A(in) * Z(in)' = Q(out) * A(out) * Z(out)'
Q(in) * B(in) * Z(in)' = Q(out) * B(out) * Z(out)'
```

ARGUMENTS

- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper triangular matrix A in the pair (A, B). On exit, the updated matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the upper triangular matrix B in the pair (A, B). On exit, the updated matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., the unitary matrix Q. On exit, the updated matrix Q. If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1; If WANTQ = .TRUE., LDQ >= N.
- **Z (input/output)**

On entry, if WANTZ = .TRUE., the unitary matrix Z. On exit, the updated matrix Z. If WANTZ = .FALSE., Z is not referenced.

- **LDZ (input)**
  The leading dimension of the array Z. LDZ > = 1; If WANTZ = .TRUE., LDZ > = N.
- **IFST (input/output)**
  Specify the reordering of the diagonal blocks of (A, B). The block with row index IFST is moved to row ILST, by a sequence of swapping between adjacent blocks.
- **ILST (input/output)**
  See the description of IFST.
- **INFO (output)**

```
 =0:  Successful exit.

 <0:  if INFO  = -i, the i-th argument had an illegal value.

 =1:  The transformed matrix pair (A, B) would be too far
from generalized Schur form; the problem is ill-
conditioned. (A, B) may have been partially reordered,
and ILST points to the first row of the current
position of the block being moved.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

```
    UMINF - 94.04, Department of Computing Science, Umea University,
    S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
    To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ztgsen - reorder the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B)

SYNOPSIS

```
 SUBROUTINE ZTGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER IWORK(*)
 LOGICAL WANTQ, WANTZ
 LOGICAL SELECT(*)
 DOUBLE PRECISION PL, PR
 DOUBLE PRECISION DIF(*)

 SUBROUTINE ZTGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, LDA, B, LDB,
*      ALPHA, BETA, Q, LDQ, Z, LDZ, M, PL, PR, DIF, WORK, LWORK, IWORK,
*      LIWORK, INFO)
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), ALPHA(*), BETA(*), Q(LDQ,*), Z(LDZ,*), WORK(*)
 INTEGER*8 IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 WANTQ, WANTZ
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION PL, PR
 DOUBLE PRECISION DIF(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSEN( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B, [LDB],
*       ALPHA, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK], [LWORK],
*       [IWORK], [LIWORK], [INFO])
 COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
 INTEGER :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
```

```
  INTEGER, DIMENSION(:) :: IWORK
  LOGICAL :: WANTQ, WANTZ
  LOGICAL, DIMENSION(:) :: SELECT
  REAL(8) :: PL, PR
  REAL(8), DIMENSION(:) :: DIF

  SUBROUTINE TGSEN_64( IJOB, WANTQ, WANTZ, SELECT, N, A, [LDA], B,
*       [LDB], ALPHA, BETA, Q, [LDQ], Z, [LDZ], M, PL, PR, DIF, [WORK],
*       [LWORK], [IWORK], [LIWORK], [INFO])
  COMPLEX(8), DIMENSION(:) :: ALPHA, BETA, WORK
  COMPLEX(8), DIMENSION(:,:) :: A, B, Q, Z
  INTEGER(8) :: IJOB, N, LDA, LDB, LDQ, LDZ, M, LWORK, LIWORK, INFO
  INTEGER(8), DIMENSION(:) :: IWORK
  LOGICAL(8) :: WANTQ, WANTZ
  LOGICAL(8), DIMENSION(:) :: SELECT
  REAL(8) :: PL, PR
  REAL(8), DIMENSION(:) :: DIF
```

C INTERFACE

#include <sunperf.h>

void ztgsen(int ijob, logical wantq, logical wantz, logical *select, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *q, int ldq, doublecomplex *z, int ldz, int *m, double *pl, double *pr, double *dif, int *info);

void ztgsen_64(long ijob, logical wantq, logical wantz, logical *select, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *alpha, doublecomplex *beta, doublecomplex *q, long ldq, doublecomplex *z, long ldz, long *m, double *pl, double *pr, double *dif, long *info);

PURPOSE

ztgsen reorders the generalized Schur decomposition of a complex matrix pair (A, B) (in terms of an unitary equivalence trans- formation Q' * (A, B) * Z), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A,B). The leading columns of Q and Z form unitary bases of the corresponding left and right eigenspaces (deflating subspaces). (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

ZTGSEN also computes the generalized eigenvalues

```
         w(j)= ALPHA(j) / BETA(j)
```

of the reordered matrix pair (A, B).

Optionally, the routine computes estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[(A11,B11), (A22,B22)] and Difl[(A11,B11), (A22,B22)], i.e. the `separation(s)` between the matrix pairs (A11, B11) and (A22,B22) that correspond to the selected cluster and the eigenvalues outside the cluster, resp., and norms of ``projections'' onto left and right eigenspaces w.r.t. the selected cluster in the (1,1)-block.

ARGUMENTS

- **IJOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (PL and PR) or the deflating subspaces (Difu and Difl):

  ```
  =0: Only reorder w.r.t. SELECT. No extras.

  =1: Reciprocal of norms of "projections" onto left and right
eigenspaces w.r.t. the selected cluster (PL and PR).
  =2: Upper bounds on Difu and Difl. F-norm-based estimate
  ```

  (DIF(1:2)).

  ```
  =3: Estimate of Difu and Difl. 1-norm-based estimate
  ```

  (DIF(1:2)). About 5 times as expensive as IJOB = 2. =4: Compute PL, PR and DIF (i.e. 0, 1 and 2 above): Economic version to get it all. =5: Compute PL, PR and DIF (i.e. 0, 1 and 3 above)
- **WANTQ (input)**
  .TRUE. : update the left transformation matrix Q;

  .FALSE.: do not update Q.
- **WANTZ (input)**
  .TRUE. : update the right transformation matrix Z;

  .FALSE.: do not update Z.
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select an eigenvalue w(j), `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrices A and B. N >= 0.
- **A (input/output)**
  On entry, the upper triangular matrix A, in generalized Schur canonical form. On exit, A is overwritten by the reordered matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input/output)**
  On entry, the upper triangular matrix B, in generalized Schur canonical form. On exit, B is overwritten by the reordered matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **ALPHA (output)**
  The diagonal elements of A and B, respectively, when the pair (A,B) has been reduced to generalized Schur form. `ALPHA(i)/BETA(i)` i =1,...,N are the generalized eigenvalues.
- **BETA (output)**
  See the description of ALPHA.
- **Q (input/output)**
  On entry, if WANTQ = .TRUE., Q is an N-by-N matrix. On exit, Q has been postmultiplied by the left unitary transformation matrix which reorder (A, B); The leading M columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTQ = .FALSE., Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ >= 1. If WANTQ = .TRUE., LDQ >= N.

- **Z (input/output)**
  On entry, if WANTZ = .TRUE., Z is an N-by-N matrix. On exit, Z has been postmultiplied by the left unitary transformation matrix which reorder (A, B); The leading M columns of Z form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces). If WANTZ = .FALSE., Z is not referenced.
- **LDZ (input)**
  The leading dimension of the array Z. LDZ >= 1. If WANTZ = .TRUE., LDZ >= N.
- **M (output)**
  The dimension of the specified pair of left and right eigenspaces, (deflating subspaces) 0 <= M <= N.
- **PL (output)**
  IF IJOB = 1, 4, or 5, PL, PR are lower bounds on the reciprocal of the norm of ``projections'' onto left and right eigenspace with respect to the selected cluster.

  0 < PL, PR <= 1. If M = 0 or M = N, PL = PR = 1. If IJOB = 0, 2, or 3 PL, PR are not referenced.
- **PR (output)**
  See the description of PL.
- **DIF (output)**
  If IJOB >= 2, DIF(1:2) store the estimates of Difu and Difl.

  If IJOB = 2 or 4, DIF(1:2) are F-norm-based upper bounds on

  Difu and Difl. If IJOB = 3 or 5, DIF(1:2) are 1-norm-based estimates of Difu and Difl, computed using reversed communication with CLACON. If M = 0 or N, DIF(1:2) = F-norm([A, B]). If IJOB = 0 or 1, DIF is not referenced.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= 1 If IJOB = 1, 2 or 4, LWORK >= 2*M*(N-M) If IJOB = 3 or 5, LWORK >= 4*M*(N-M)

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced. Otherwise, on exit, if INFO = 0, IWORK(1) returns the optimal LIWORK.
- **LIWORK (input)**
  The dimension of the array IWORK. LIWORK >= 1. If IJOB = 1, 2 or 4, LIWORK >= N+2; If IJOB = 3 or 5, LIWORK >= MAX(N+2, 2*M*(N-M));

  If LIWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the IWORK array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by XERBLA.
- **INFO (output)**

```
 =0: Successful exit.

 <0: If INFO  = -i, the i-th argument had an illegal value.

 =1: Reordering of (A, B) failed because the transformed
matrix pair (A, B) would be too far from generalized
Schur form; the problem is very ill-conditioned.
(A, B) may have been partially reordered.
If requested, 0 is returned in DIF(*), PL and PR.
```

FURTHER DETAILS

ZTGSEN first collects the selected eigenvalues by computing unitary U and W that move them to the top left corner of (A, B). In other words, the selected eigenvalues are the eigenvalues of (A11, B11) in

```
U'*(A, B)*W  = (A11 A12) (B11 B12) n1
               ( 0  A22),( 0  B22) n2
                 n1  n2    n1  n2
```

where N = n1+n2 and U' means the conjugate transpose of U. The first n1 columns of U and W span the specified pair of left and right eigenspaces (deflating subspaces) of (A, B).

If (A, B) has been obtained from the generalized real Schur decomposition of a matrix pair (C, D) = Q*(A, B)*Z', then the reordered generalized Schur form of (C, D) is given by

```
(C, D)  = (Q*U)*(U'*(A, B)*W)*(Z*W)',
```

and the first n1 columns of Q*U and Z*W span the corresponding deflating subspaces of (C, D) (Q and Z store Q*U and Z*W, resp.).

Note that if the selected eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

The reciprocal condition numbers of the left and right eigenspaces spanned by the first n1 columns of U and W (or Q*U and Z*W) may be returned in DIF(1:2), corresponding to Difu and Difl, resp.

The Difu and Difl are defined as:

ifu[(A11, B11), (A22, B22)] = sigma-min(Zu)

and ifl[(A11, B11), (A22, B22)] = Difu[(A22, B22), (A11, B11)],

where sigma-min(Zu) is the smallest singular value of the (2*n1*n2)-by-(2*n1*n2) matrix

u = [kron(In2, A11) -kron(A22', In1)]

```
    [ kron(In2, B11)  -kron(B22', In1) ].
```

Here, Inx is the identity matrix of size nx and A22' is the transpose of A22. kron(X, Y) is the Kronecker product between the matrices X and Y.

When DIF(2) is small, small changes in (A, B) can cause large changes in the deflating subspace. An approximate (asymptotic) bound on the maximum angular error in the computed deflating subspaces is PS * norm((A, B)) / DIF(2),

where EPS is the machine precision.

The reciprocal norm of the projectors on the left and right eigenspaces associated with (A11, B11) may be returned in PL and PR. They are computed as follows. First we compute L and R so that P*(A, B)*Q is block diagonal, where

```
= ( I -L ) n1          Q  = ( I R ) n1
  ( 0  I ) n2    and         ( 0 I ) n2
    n1 n2                     n1 n2
```

and (L, R) is the solution to the generalized Sylvester equation 11*R - L*A22 = -A12 11*R - L*B22 = -B12

Then PL = (F-norm(L)**2+1)**(-1/2) and PR = (F-norm(R)**2+1)**(-1/2). An approximate (asymptotic) bound on the average absolute error of the selected eigenvalues is

EPS * norm((A, B)) / PL.

There are also global error bounds which valid for perturbations up to a certain restriction: A lower bound (x) on the smallest F-norm(E,F) for which an eigenvalue of (A11, B11) may move and coalesce with an eigenvalue of (A22, B22) under perturbation (E,F), (i.e. (A + E, B + F), is

```
x  = min(Difu,Difl)/((1/(PL*PL)+1/(PR*PR))**(1/2)+2*max(1/PL,1/PR)).
```

An approximate bound on x can be computed from DIF(1:2), PL and PR.

If y = (F-norm(E,F) / x) < = 1, the angles between the perturbed (L', R') and unperturbed (L, R) left and right deflating subspaces associated with the selected cluster in the (1,1)-blocks can be bounded as

```
max-angle(L, L')  < = arctan( y * PL / (1 - y * (1 - PL * PL)**(1/2))
max-angle(R, R')  < = arctan( y * PR / (1 - y * (1 - PR * PR)**(1/2))
```

See LAPACK User's Guide section 4.11 or the following references for more information.

Note that if the default method for computing the Frobenius-norm- based estimate DIF is not wanted (see CLATDF), then the parameter IDIFJB (see below) should be changed from 3 to 4 (routine CLATDF (IJOB = 2 will be used)). See CTGSYL for more details.

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

References

```
 = = = = = = = = = =
```

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

```
    UMINF - 94.04, Department of Computing Science, Umea University,
    S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
    To appear in Numerical Algorithms, 1996.
```

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ztgsja - compute the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B

SYNOPSIS

```
 SUBROUTINE ZTGSJA( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B, LDB,
*      TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK, NCYCLE,
*      INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION ALPHA(*), BETA(*)

 SUBROUTINE ZTGSJA_64( JOBU, JOBV, JOBQ, M, P, N, K, L, A, LDA, B,
*      LDB, TOLA, TOLB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ, WORK,
*      NCYCLE, INFO)
 CHARACTER * 1 JOBU, JOBV, JOBQ
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), U(LDU,*), V(LDV,*), Q(LDQ,*), WORK(*)
 INTEGER*8 M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 DOUBLE PRECISION TOLA, TOLB
 DOUBLE PRECISION ALPHA(*), BETA(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSJA( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: ALPHA, BETA

 SUBROUTINE TGSJA_64( JOBU, JOBV, JOBQ, [M], [P], [N], K, L, A, [LDA],
*       B, [LDB], TOLA, TOLB, ALPHA, BETA, U, [LDU], V, [LDV], Q, [LDQ],
```

```
*       [WORK], NCYCLE, [INFO])
 CHARACTER(LEN=1) :: JOBU, JOBV, JOBQ
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, U, V, Q
 INTEGER(8) :: M, P, N, K, L, LDA, LDB, LDU, LDV, LDQ, NCYCLE, INFO
 REAL(8) :: TOLA, TOLB
 REAL(8), DIMENSION(:) :: ALPHA, BETA
```

C INTERFACE

#include <sunperf.h>

void ztgsja(char jobu, char jobv, char jobq, int m, int p, int n, int k, int l, doublecomplex *a, int lda, doublecomplex *b, int ldb, double tola, double tolb, double *alpha, double *beta, doublecomplex *u, int ldu, doublecomplex *v, int ldv, doublecomplex *q, int ldq, int *ncycle, int *info);

void ztgsja_64(char jobu, char jobv, char jobq, long m, long p, long n, long k, long l, doublecomplex *a, long lda, doublecomplex *b, long ldb, double tola, double tolb, double *alpha, double *beta, doublecomplex *u, long ldu, doublecomplex *v, long ldv, doublecomplex *q, long ldq, long *ncycle, long *info);

PURPOSE

ztgsja computes the generalized singular value decomposition (GSVD) of two complex upper triangular (or trapezoidal) matrices A and B.

On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine CGGSVP from a general M-by-N matrix A and P-by-N matrix B:

```
             N-K-L  K    L

   A =    K ( 0    A12  A13 ) if M-K-L >= 0;

          L ( 0     0   A23 )

      M-K-L ( 0     0    0  )

           N-K-L  K    L

   A =  K ( 0    A12  A13 ) if M-K-L < 0;

      M-K ( 0     0   A23 )

           N-K-L  K    L

   B =  L ( 0     0   B13 )

      P-L ( 0     0    0  )
```

where the K-by-K matrix A12 and L-by-L matrix B13 are nonsingular upper triangular; A23 is L-by-L upper triangular if M-K-L >= 0, otherwise A23 is (M-K)-by-L upper trapezoidal.

On exit,

```
       U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R ),
```

where U, V and Q are unitary matrices, Z' denotes the conjugate transpose of Z, R is a nonsingular upper triangular matrix, and D1 and D2 are ``diagonal'' matrices, which are of the following structures:

If M-K-L >= 0,

```
                    K  L
       D1 =     K ( I  0 )
                L ( 0  C )
            M-K-L ( 0  0 )

                   K  L
       D2 = L   ( 0  S )
            P-L ( 0  0 )

               N-K-L  K    L
  ( 0 R ) = K (  0   R11  R12 ) K
            L (  0    0   R22 ) L
```

where

```
  C = diag( ALPHA(K+1), ... , ALPHA(K+L) ),
  S = diag( BETA(K+1),  ... , BETA(K+L) ),
  C**2 + S**2 = I.
  R is stored in A(1:K+L,N-K-L+1:N) on exit.
```

If M-K-L < 0,

```
                 K M-K K+L-M
    D1 =   K ( I  0    0   )
         M-K ( 0  C    0   )
                   K M-K K+L-M
    D2 =   M-K ( 0  S    0   )
         K+L-M ( 0  0    I   )
```

```
          P-L ( 0  0    0   )

               N-K-L  K   M-K  K+L-M

         M-K ( 0     0   R22  R23  )

       K+L-M ( 0     0    0   R33  )
```

where

C = diag(ALPHA(K+1), ... , ALPHA(M)),

S = diag(BETA(K+1), ... , BETA(M)),

C**2 + S**2 = I.

R = (R11 R12 R13) is stored in A(1:M, N-K-L+1:N) and R33 is stored (0 R22 R23)

in B(M-K+1:L,N+M-K-L+1:N) on exit.

The computation of the unitary transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U1, V1, or Q1.

CTGSJA essentially uses a variant of Kogbetliantz algorithm to reduce min(L,M-K)-by-L triangular (or trapezoidal) matrix A23 and L-by-L matrix B13 to the form:

```
U1'*A13*Q1 = C1*R1; V1'*B13*Q1 = S1*R1,
```

where U1, V1 and Q1 are unitary matrix, and Z' is the conjugate transpose of Z. C1 and S1 are diagonal matrices satisfying

```
C1**2 + S1**2 = I,
```

and R1 is an L-by-L nonsingular upper triangular matrix.

ARGUMENTS

- **JOBU (input)**

```
 = 'U':  U must contain a unitary matrix U1 on entry, and
the product U1*U is returned;
 = 'I':  U is initialized to the unit matrix, and the
unitary matrix U is returned;
 = 'N':  U is not computed.
```

- **JOBV (input)**

```
 = 'V':  V must contain a unitary matrix V1 on entry, and
the product V1*V is returned;
 = 'I':  V is initialized to the unit matrix, and the
unitary matrix V is returned;
 = 'N':  V is not computed.
```

- **JOBQ (input)**

```
 = 'Q':  Q must contain a unitary matrix Q1 on entry, and
the product Q1*Q is returned;
 = 'I':  Q is initialized to the unit matrix, and the
unitary matrix Q is returned;
 = 'N':  Q is not computed.
```

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **P (input)**
  The number of rows of the matrix B. P >= 0.
- **N (input)**
  The number of columns of the matrices A and B. N >= 0.
- **K (input)**
  K and L specify the subblocks in the input matrices A and B:

  A23 = A(K+1:MIN(K+L,M),N-L+1:N) and B13 = B(1:L,,N-L+1:N) of A and B, whose GSVD is going to be computed by CTGSJA. See the Further Details section below.
- **L (input)**
  See the description of K.
- **A (input/output)**
  On entry, the M-by-N matrix A. On exit, A(N-K+1:N,1:MIN(K+L,M)) contains the triangular matrix R or part of R. See Purpose for details.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input/output)**
  On entry, the P-by-N matrix B. On exit, if necessary, B(M-K+1:L,N+M-K-L+1:N) contains a part of R. See Purpose for details.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,P).
- **TOLA (input)**
  TOLA and TOLB are the convergence criteria for the Jacobi- Kogbetliantz iteration procedure. Generally, they are the same as used in the preprocessing step, say TOLA = MAX(M,N)*norm(A)*MACHEPS, TOLB = MAX(P,N)*norm(B)*MACHEPS.
- **TOLB (input)**
  See the description of TOLA.
- **ALPHA (output)**
  On exit, ALPHA and BETA contain the generalized singular value pairs of A and B; ALPHA(1:K) = 1,

  BETA(1:K) = 0, and if M-K-L >= 0, ALPHA(K+1:K+L) = diag(C),

  BETA(K+1:K+L) = diag(S), or if M-K-L < 0, ALPHA(K+1:M) = C, ALPHA(M+1:K+L) = 0

  BETA(K+1:M) = S, BETA(M+1:K+L) = 1. Furthermore, if K+L < N, ALPHA(K+L+1:N) = 0

  BETA(K+L+1:N) = 0.
- **BETA (output)**
  See the description of ALPHA.
- **U (input/output)**
  On entry, if JOBU = 'U', U must contain a matrix U1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBU = 'I', U contains the unitary matrix U; if JOBU = 'U', U contains the product U1*U. If JOBU = 'N', U is not referenced.
- **LDU (input)**
  The leading dimension of the array U. LDU >= max(1,M) if JOBU = 'U'; LDU >= 1 otherwise.

- **V (input/output)**
  On entry, if JOBV = 'V', V must contain a matrix V1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBV = 'I', V contains the unitary matrix V; if JOBV = 'V', V contains the product V1*V. If JOBV = 'N', V is not referenced.
- **LDV (input)**
  The leading dimension of the array V. LDV > = `max(1,P)` if JOBV = 'V'; LDV > = 1 otherwise.
- **Q (input/output)**
  On entry, if JOBQ = 'Q', Q must contain a matrix Q1 (usually the unitary matrix returned by CGGSVP). On exit, if JOBQ = 'I', Q contains the unitary matrix Q; if JOBQ = 'Q', Q contains the product Q1*Q. If JOBQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = `max(1,N)` if JOBQ = 'Q'; LDQ > = 1 otherwise.
- **WORK (workspace)**
  `dimension(2*N)`
- **NCYCLE (output)**
  The number of cycles required for convergence.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value.

= 1:  the procedure does not converge after MAXIT cycles.
```

NAME

ztgsna - estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B)

SYNOPSIS

```
 SUBROUTINE ZTGSNA( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL, LDVL,
*      VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER IWORK(*)
 LOGICAL SELECT(*)
 DOUBLE PRECISION S(*), DIF(*)

 SUBROUTINE ZTGSNA_64( JOB, HOWMNT, SELECT, N, A, LDA, B, LDB, VL,
*      LDVL, VR, LDVR, S, DIF, MM, M, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 JOB, HOWMNT
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER*8 IWORK(*)
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION S(*), DIF(*)
```

F95 INTERFACE

```
 SUBROUTINE TGSNA( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB], VL,
*       [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, DIF
```

```
 SUBROUTINE TGSNA_64( JOB, HOWMNT, SELECT, [N], A, [LDA], B, [LDB],
*       VL, [LDVL], VR, [LDVR], S, DIF, MM, M, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNT
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, VL, VR
 INTEGER(8) :: N, LDA, LDB, LDVL, LDVR, MM, M, LWORK, INFO
 INTEGER(8), DIMENSION(:) :: IWORK
 LOGICAL(8), DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, DIF
```

C INTERFACE

#include <sunperf.h>

void ztgsna(char job, char howmnt, logical *select, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, double *s, double *dif, int mm, int *m, int *info);

void ztgsna_64(char job, char howmnt, logical *select, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, double *s, double *dif, long mm, long *m, long *info);

PURPOSE

ztgsna estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B).

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (DIF):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (DIF);

  = 'B': for both eigenvalues and eigenvectors (S and DIF).
  ```
- **HOWMNT (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNT = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the corresponding j-th eigenvalue and/or eigenvector, `SELECT(j)` must be set to .TRUE.. If HOWMNT = 'A', SELECT is not referenced.
- **N (input)**

The order of the square matrix pair (A, B). N > = 0.

- **A (input)**
  The upper triangular matrix A in the pair (A,B).
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **B (input)**
  The upper triangular matrix B in the pair (A, B).
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by CTGEVC. If JOB = 'V', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and If JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of (A, B), corresponding to the eigenpairs specified by HOWMNT and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by CTGEVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; If JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If JOB = 'V', S is not referenced.
- **DIF (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute DIF(j), `DIF(j)` is set to 0; this can only occur when the true value would be very small anyway. For each eigenvalue/vector specified by SELECT, DIF stores a Frobenius norm-based estimate of Difl. If JOB = 'E', DIF is not referenced.
- **MM (input)**
  The number of elements in the arrays S and DIF. MM > = M.
- **M (output)**
  The number of elements of the arrays S and DIF used to store the specified condition numbers; for each selected eigenvalue one element is used. If HOWMNT = 'A', M is set to N.
- **WORK (workspace)**
  If JOB = 'E', WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. If JOB = 'V' or 'B', LWORK > = 2*N*N.
- **IWORK (workspace)**
  `dimension(N+2)` If JOB = 'E', IWORK is not referenced.
- **INFO (output)**

```
= 0: Successful exit

< 0: If INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of the i-th generalized eigenvalue w = (a, b) is defined as

```
S(I)  = (|v'Au|**2 + |v'Bu|**2)**(1/2) / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of (A, B) corresponding to w; |z| denotes the absolute value of the complex number, and `norm(u)` denotes the 2-norm of the vector u. The pair (a, b) corresponds to an eigenvalue w = a/b (= v'Au/v'Bu) of the matrix pair (A, B). If both a and b equal zero, then (A,B) is singular and `S(I)` = -1 is returned.

An approximate error bound on the chordal distance between the i-th computed generalized eigenvalue w and the corresponding exact eigenvalue lambda is

```
chord(w, lambda)  < =   EPS * norm(A, B) / S(I),
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u and left eigenvector v corresponding to the generalized eigenvalue w is defined as follows. Suppose

```
(A, B)  = ( a   *  ) ( b  *  )  1

          ( 0  A22 ),( 0 B22 )  n-1

            1  n-1     1 n-1
```

Then the reciprocal condition number `DIF(I)` is

```
Difl[(a, b), (A22, B22)]   = sigma-min( Zl )
```

where sigma-min(Zl) denotes the smallest singular value of

```
Zl  = [ kron(a, In-1) -kron(1, A22) ]

      [ kron(b, In-1) -kron(1, B22) ].
```

Here In-1 is the identity matrix of size n-1 and X' is the conjugate transpose of X. kron(X, Y) is the Kronecker product between the matrices X and Y.

We approximate the smallest singular value of Zl with an upper bound. This is done by CLATDF.

An approximate error bound for a computed eigenvector `VL(i)` or `VR(i)` is given by

```
EPS * norm(A, B) / DIF(i).
```

See ref. [2-3] for more details and further references.

Based on contributions by


```
Bo Kagstrom and Peter Poromaa, Department of Computing Science,
Umea University, S-901 87 Umea, Sweden.
```


References

= = = = = = = = = =

[1] B. Kagstrom; A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A, B), in M.S. Moonen et al (eds), Linear Algebra for Large Scale and Real-Time Applications, Kluwer Academic Publ. 1993, pp 195-218.

[2] B. Kagstrom and P. Poromaa; Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software, Report

UMINF - 94.04, Department of Computing Science, Umea University,
S-901 87 Umea, Sweden, 1994. Also as LAPACK Working Note 87.
To appear in Numerical Algorithms, 1996.

[3] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75.

To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

NAME

ztgsyl - solve the generalized Sylvester equation

SYNOPSIS

```
 SUBROUTINE ZTGSYL( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
 INTEGER IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER IWORK(*)
 DOUBLE PRECISION SCALE, DIF

 SUBROUTINE ZTGSYL_64( TRANS, IJOB, M, N, A, LDA, B, LDB, C, LDC, D,
*      LDD, E, LDE, F, LDF, SCALE, DIF, WORK, LWORK, IWORK, INFO)
 CHARACTER * 1 TRANS
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*), D(LDD,*), E(LDE,*), F(LDF,*), WORK(*)
 INTEGER*8 IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER*8 IWORK(*)
 DOUBLE PRECISION SCALE, DIF
```

F95 INTERFACE

```
 SUBROUTINE TGSYL( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C, [LDC],
*       D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK], [IWORK],
*       [INFO])
 CHARACTER(LEN=1) :: TRANS
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, C, D, E, F
 INTEGER :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
 INTEGER, DIMENSION(:) :: IWORK
 REAL(8) :: SCALE, DIF

 SUBROUTINE TGSYL_64( TRANS, IJOB, [M], [N], A, [LDA], B, [LDB], C,
*       [LDC], D, [LDD], E, [LDE], F, [LDF], SCALE, DIF, [WORK], [LWORK],
*       [IWORK], [INFO])
```

```
CHARACTER(LEN=1) :: TRANS
COMPLEX(8), DIMENSION(:) :: WORK
COMPLEX(8), DIMENSION(:,:) :: A, B, C, D, E, F
INTEGER(8) :: IJOB, M, N, LDA, LDB, LDC, LDD, LDE, LDF, LWORK, INFO
INTEGER(8), DIMENSION(:) :: IWORK
REAL(8) :: SCALE, DIF
```

C INTERFACE

#include <sunperf.h>

void ztgsyl(char trans, int ijob, int m, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *c, int ldc, doublecomplex *d, int ldd, doublecomplex *e, int lde, doublecomplex *f, int ldf, double *scale, double *dif, int *info);

void ztgsyl_64(char trans, long ijob, long m, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *c, long ldc, doublecomplex *d, long ldd, doublecomplex *e, long lde, doublecomplex *f, long ldf, double *scale, double *dif, long *info);

PURPOSE

ztgsyl solves the generalized Sylvester equation:

```
A * R - L * B = scale * C            (1)

D * R - L * E = scale * F
```

where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively, with complex entries. A, B, D and E are upper triangular (i.e., (A,D) and (B,E) in generalized Schur form).

The solution (R, L) overwrites (C, F). 0 <= SCALE <= 1

is an output scaling factor chosen to avoid overflow.

In matrix notation (1) is equivalent to solve Zx = scale*b, where Z is defined as

```
Z = [ kron(In, A)  -kron(B', Im) ]        (2)

    [ kron(In, D)  -kron(E', Im) ],
```

Here Ix is the identity matrix of size x and X' is the conjugate transpose of X. Kron(X, Y) is the Kronecker product between the matrices X and Y.

If TRANS = 'C', y in the conjugate transposed system Z'*y = scale*b is solved for, which is equivalent to solve for R and L in

```
A' * R + D' * L = scale * C           (3)

R * B' + L * E' = scale * -F
```

This case (TRANS = 'C') is used to compute an one-norm-based estimate of Dif[(A,D), (B,E)], the separation between the matrix pairs (A,D) and (B,E), using CLACON.

If IJOB >= 1, CTGSYL computes a Frobenius norm-based estimate of Dif[(A,D),(B,E)]. That is, the reciprocal of a lower

bound on the reciprocal of the smallest singular value of Z.

This is a level-3 BLAS algorithm.

ARGUMENTS

- **TRANS (input)**

```
= 'N': solve the generalized sylvester equation (1).

= 'C': solve the "conjugate transposed" system (3).
```

- **IJOB (input)**
  Specifies what kind of functionality to be performed. =0: solve (1) only.

```
=1: The functionality of 0 and 3.

=2: The functionality of 0 and 4.

=3: Only an estimate of Dif[(A,D), (B,E)] is computed.
(look ahead strategy is used).
=4: Only an estimate of Dif[(A,D), (B,E)] is computed.
(CGECON on sub-systems is used).
Not referenced if TRANS  = 'C'.
```

- **M (input)**
  The order of the matrices A and D, and the row dimension of the matrices C, F, R and L.
- **N (input)**
  The order of the matrices B and E, and the column dimension of the matrices C, F, R and L.
- **A (input)**
  The upper triangular matrix A.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1, M).
- **B (input)**
  The upper triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1, N).
- **C (input/output)**
  On entry, C contains the right-hand-side of the first matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, C has been overwritten by the solution R. If IJOB = 3 or 4 and TRANS = 'N', C holds R, the solution achieved during the computation of the Dif-estimate.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1, M).
- **D (input)**
  The upper triangular matrix D.
- **LDD (input)**
  The leading dimension of the array D. LDD >= max(1, M).
- **E (input)**
  The upper triangular matrix E.
- **LDE (input)**
  The leading dimension of the array E. LDE >= max(1, N).

- **F (input/output)**
  On entry, F contains the right-hand-side of the second matrix equation in (1) or (3). On exit, if IJOB = 0, 1 or 2, F has been overwritten by the solution L. If IJOB = 3 or 4 and TRANS = 'N', F holds L, the solution achieved during the computation of the Dif-estimate.
- **LDF (input)**
  The leading dimension of the array F. LDF > = max(1, M).
- **SCALE (output)**
  On exit SCALE is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. SCALE is an upper bound of Dif[(A,D), (B,E)] = sigma-min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'C', SCALE is not referenced.
- **DIF (output)**
  On exit SCALE is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. SCALE is an upper bound of Dif[(A,D), (B,E)] = sigma-min(Z), where Z as in (2). If IJOB = 0 or TRANS = 'C', SCALE is not referenced.
- **WORK (workspace)**
  If IJOB = 0, WORK is not referenced. Otherwise, on exit, if INFO =0 then WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = 1. If IJOB = 1 or 2 and TRANS = 'N', LWORK > = 2*M*N.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **IWORK (workspace)**
  If IJOB = 0, IWORK is not referenced.
- **INFO (output)**

```
 =0: successful exit

 <0: If INFO  = -i, the i-th argument had an illegal value.

 >0: (A, D) and (B, E) have common or very close
eigenvalues.
```

FURTHER DETAILS

Based on contributions by

```
   Bo Kagstrom and Peter Poromaa, Department of Computing Science,
   Umea University, S-901 87 Umea, Sweden.
```

[1] B. Kagstrom and P. Poromaa, LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs, Report UMINF - 93.23, Department of Computing Science, Umea University, S-901 87 Umea, Sweden, December 1993, Revised April 1994, Also as LAPACK Working Note 75. To appear in ACM Trans. on Math. Software, Vol 22, No 1, 1996.

[2] B. Kagstrom, A Perturbation Analysis of the Generalized Sylvester Equation (AR - LB, DR - LE) = (C, F), SIAM J. Matrix Anal. Appl., 15(4):1045-1060, 1994.

[3] B. Kagstrom and L. Westin, Generalized Schur Methods with Condition Estimators for Solving the Generalized Sylvester Equation, IEEE Transactions on Automatic Control, Vol. 34, No. 7, July 1989, pp 745-751.

NAME

ztpcon - estimate the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE ZTPCON( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(*), WORK(*)
 INTEGER N, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZTPCON_64( NORM, UPLO, DIAG, N, A, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(*), WORK(*)
 INTEGER*8 N, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TPCON( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: A, WORK
 INTEGER :: N, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE TPCON_64( NORM, UPLO, DIAG, N, A, RCOND, [WORK], [WORK2],
*       [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: A, WORK
 INTEGER(8) :: N, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ztpcon(char norm, char uplo, char diag, int n, doublecomplex *a, double *rcond, int *info);

void ztpcon_64(char norm, char uplo, char diag, long n, doublecomplex *a, double *rcond, long *info);

PURPOSE

ztpcon estimates the reciprocal of the condition number of a packed triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.
- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).

- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

  `= 0:  successful exit`

  `< 0:  if INFO  = -i, the i-th argument had an illegal value`

NAME

ztpmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE ZTPMV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), Y(*)
INTEGER N, INCY

SUBROUTINE ZTPMV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE TPMV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A, Y
INTEGER :: N, INCY

SUBROUTINE TPMV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A, Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void ztpmv(char uplo, char transa, char diag, int n, doublecomplex *a, doublecomplex *y, int incy);

void ztpmv_64(char uplo, char transa, char diag, long n, doublecomplex *a, doublecomplex *y, long incy);

PURPOSE

ztpmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztprfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE ZTPRFS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X, LDX,
*       FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZTPRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, X,
*       LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TPRFS( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB], X,
*        [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: A, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TPRFS_64( UPLO, [TRANSA], DIAG, N, [NRHS], A, B, [LDB],
*        X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: A, WORK
 COMPLEX(8), DIMENSION(:,:) :: B, X
 INTEGER(8) :: N, NRHS, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ztprfs(char uplo, char transa, char diag, int n, int nrhs, doublecomplex *a, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void ztprfs_64(char uplo, char transa, char diag, long n, long nrhs, doublecomplex *a, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

ztprfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular packed coefficient matrix.

The solution matrix X must be computed by CTPTRS or some other means before entering this routine. CTPRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2n-j)/2) = `A(i,j)` for j < =i < =n. If DIAG = 'U', the diagonal elements of A are not referenced and are assumed to be 1.

- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX > = max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztpsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE ZTPSV( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), Y(*)
INTEGER N, INCY

SUBROUTINE ZTPSV_64( UPLO, TRANSA, DIAG, N, A, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), Y(*)
INTEGER*8 N, INCY
```

F95 INTERFACE

```
SUBROUTINE TPSV( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A, Y
INTEGER :: N, INCY

SUBROUTINE TPSV_64( UPLO, [TRANSA], DIAG, [N], A, Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A, Y
INTEGER(8) :: N, INCY
```

C INTERFACE

#include <sunperf.h>

void ztpsv(char uplo, char transa, char diag, int n, doublecomplex *a, doublecomplex *y, int incy);

void ztpsv_64(char uplo, char transa, char diag, long n, doublecomplex *a, doublecomplex *y, long incy);

PURPOSE

ztpsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  ((n*(n + 1))/2). Before entry with UPLO = 'U' or 'u', the array A must contain the upper triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(1, 2) and a(2, 2) respectively, and so on. Before entry with UPLO = 'L' or 'l', the array A must contain the lower triangular matrix packed sequentially, column by column, so that A(1) contains a(1, 1), A(2) and A(3) contain a(2, 1) and a(3, 1) respectively, and so on. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity. Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztptri - compute the inverse of a complex upper or lower triangular matrix A stored in packed format

---

SYNOPSIS

```
SUBROUTINE ZTPTRI( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(*)
INTEGER N, INFO

SUBROUTINE ZTPTRI_64( UPLO, DIAG, N, A, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(*)
INTEGER*8 N, INFO
```

F95 INTERFACE

```
SUBROUTINE TPTRI( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:) :: A
INTEGER :: N, INFO

SUBROUTINE TPTRI_64( UPLO, DIAG, N, A, [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:) :: A
INTEGER(8) :: N, INFO
```

C INTERFACE

#include <sunperf.h>

void ztptri(char uplo, char diag, int n, doublecomplex *a, int *info);

void ztptri_64(char uplo, char diag, long n, doublecomplex *a, long *info);

PURPOSE

ztptri computes the inverse of a complex upper or lower triangular matrix A stored in packed format.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the upper or lower triangular matrix A, stored columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = A(i,j) for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*((2*n-j)/2) = A(i,j) for j < =i < =n. See below for further details. On exit, the (triangular) inverse of the original matrix, in the same packed storage format.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value

 > 0:  if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

FURTHER DETAILS

A triangular matrix A can be transferred to packed storage using one of the following program segments:

UPLO = 'U': UPLO = 'L':

```
JC  = 1                          JC  = 1

DO 2 J  = 1, N                   DO 2 J  = 1, N
```

```
     DO 1 I  = 1, J
        A(JC+I-1)  = A(I,J)
1    CONTINUE
     JC  = JC + J
2 CONTINUE
```

```
      DO 1 I  = J, N
        A(JC+I-J)  = A(I,J)
1    CONTINUE
      JC  = JC + N - J + 1
2 CONTINUE
```

NAME

ztptrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
SUBROUTINE ZTPTRS( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER N, NRHS, LDB, INFO

SUBROUTINE ZTPTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, B, LDB, INFO)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(*), B(LDB,*)
INTEGER*8 N, NRHS, LDB, INFO
```

F95 INTERFACE

```
SUBROUTINE TPTRS( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB], [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER :: N, NRHS, LDB, INFO

SUBROUTINE TPTRS_64( UPLO, TRANSA, DIAG, N, [NRHS], A, B, [LDB],
*       [INFO])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: A
COMPLEX(8), DIMENSION(:,:) :: B
INTEGER(8) :: N, NRHS, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ztptrs(char uplo, char transa, char diag, int n, int nrhs, doublecomplex *a, doublecomplex *b, int ldb, int *info);

void ztptrs_64(char uplo, char transa, char diag, long n, long nrhs, doublecomplex *a, doublecomplex *b, long ldb, long *info);

PURPOSE

ztptrs solves a triangular system of the form

where A is a triangular matrix of order N stored in packed format, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **TRANSA (input)**
  Specifies the form of the system of equations:

```
= 'N':  A * X  = B     (No transpose)

= 'T':  A**T * X  = B  (Transpose)

= 'C':  A**H * X  = B  (Conjugate transpose)
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.
- **A (input)**
  The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array A as follows: if UPLO = 'U', A(i + (j-1)*j/2) = `A(i,j)` for 1 < =i < =j; if UPLO = 'L', A(i + (j-1)*(2*n-j)/2) = `A(i,j)` for j < =i < =n.
- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
= 0:  successful exit
```

 < 0: if INFO = -i, the i-th argument had an illegal value

 > 0: if INFO = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the
solutions X have not been computed.

NAME

ztrans - transpose and scale source matrix

SYNOPSIS

```
SUBROUTINE ZTRANS( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
DOUBLE COMPLEX SCALE
DOUBLE COMPLEX SOURCE(*), DEST(*)
INTEGER M, N

SUBROUTINE ZTRANS_64( PLACE, SCALE, SOURCE, M, N, DEST)
CHARACTER * 1 PLACE
DOUBLE COMPLEX SCALE
DOUBLE COMPLEX SOURCE(*), DEST(*)
INTEGER*8 M, N
```

F95 INTERFACE

```
SUBROUTINE TRANS( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
COMPLEX(8) :: SCALE
COMPLEX(8), DIMENSION(:) :: SOURCE, DEST
INTEGER :: M, N

SUBROUTINE TRANS_64( [PLACE], SCALE, SOURCE, M, N, DEST)
CHARACTER(LEN=1) :: PLACE
COMPLEX(8) :: SCALE
COMPLEX(8), DIMENSION(:) :: SOURCE, DEST
INTEGER(8) :: M, N
```

C INTERFACE

#include <sunperf.h>

void ztrans(char place, doublecomplex scale, doublecomplex *source, int m, int n, doublecomplex *dest);

void ztrans_64(char place, doublecomplex scale, doublecomplex *source, long m, long n, doublecomplex *dest);

PURPOSE

ztrans scales and transposes the source matrix. The N2 x N1 result is written into SOURCE when PLACE = 'I' or 'i', and DEST when PLACE = 'O' or 'o'.

```
PLACE = 'I' or 'i': SOURCE = SCALE * SOURCE'

PLACE = 'O' or 'o': DEST = SCALE * SOURCE'
```

ARGUMENTS

- **PLACE (input)**
  Type of transpose. 'I' or 'i' for in-place, 'O' or 'o' for out-of-place. 'I' is default.
- **SCALE (input)**
  Scale factor on the SOURCE matrix.
- **SOURCE (input/output)**
  on input. Array of (N, M) on output if in-place transpose.
- **M (input)**
  Number of rows in the SOURCE matrix on input.
- **N (input)**
  Number of columns in the SOURCE matrix on input.
- **DEST (output)**
  Scaled and transposed SOURCE matrix if out-of-place transpose. Not referenced if in-place transpose.

NAME

ztrcon - estimate the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm

---

SYNOPSIS

```
 SUBROUTINE ZTRCON( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK, WORK2,
*      INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER N, LDA, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)

 SUBROUTINE ZTRCON_64( NORM, UPLO, DIAG, N, A, LDA, RCOND, WORK,
*      WORK2, INFO)
 CHARACTER * 1 NORM, UPLO, DIAG
 DOUBLE COMPLEX A(LDA,*), WORK(*)
 INTEGER*8 N, LDA, INFO
 DOUBLE PRECISION RCOND
 DOUBLE PRECISION WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TRCON( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER :: N, LDA, INFO
 REAL(8) :: RCOND
 REAL(8), DIMENSION(:) :: WORK2

 SUBROUTINE TRCON_64( NORM, UPLO, DIAG, [N], A, [LDA], RCOND, [WORK],
*       [WORK2], [INFO])
 CHARACTER(LEN=1) :: NORM, UPLO, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A
 INTEGER(8) :: N, LDA, INFO
```

```
REAL(8) :: RCOND
REAL(8), DIMENSION(:) :: WORK2
```

C INTERFACE

#include <sunperf.h>

void ztrcon(char norm, char uplo, char diag, int n, doublecomplex *a, int lda, double *rcond, int *info);

void ztrcon_64(char norm, char uplo, char diag, long n, doublecomplex *a, long lda, double *rcond, long *info);

PURPOSE

ztrcon estimates the reciprocal of the condition number of a triangular matrix A, in either the 1-norm or the infinity-norm.

The norm of A is computed and an estimate is obtained for norm(inv(A)), then the reciprocal of the condition number is computed as

```
RCOND = 1 / ( norm(A) * norm(inv(A)) ).
```

ARGUMENTS

- **NORM (input)**
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required:

  ```
  = '1' or 'O':  1-norm;

  = 'I':         Infinity-norm.
  ```
- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```
- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,N).

- **RCOND (output)**
  The reciprocal of the condition number of the matrix A, computed as RCOND = 1/(norm(A) * norm(inv(A))).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztrevc - compute some or all of the right and/or left eigenvectors of a complex upper triangular matrix T

SYNOPSIS

```
 SUBROUTINE ZTREVC( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 DOUBLE COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION RWORK(*)

 SUBROUTINE ZTREVC_64( SIDE, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*       LDVR, MM, M, WORK, RWORK, INFO)
 CHARACTER * 1 SIDE, HOWMNY
 DOUBLE COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(*)
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION RWORK(*)
```

F95 INTERFACE

```
 SUBROUTINE TREVC( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: T, VL, VR
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: RWORK

 SUBROUTINE TREVC_64( SIDE, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], MM, M, [WORK], [RWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, HOWMNY
 COMPLEX(8), DIMENSION(:) :: WORK
```

```
COMPLEX(8), DIMENSION(:,:) :: T, VL, VR
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8), DIMENSION(:) :: RWORK
```

C INTERFACE

#include <sunperf.h>

void ztrevc(char side, char howmny, logical *select, int n, doublecomplex *t, int ldt, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, int mm, int *m, int *info);

void ztrevc_64(char side, char howmny, logical *select, long n, doublecomplex *t, long ldt, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, long mm, long *m, long *info);

PURPOSE

ztrevc computes some or all of the right and/or left eigenvectors of a complex upper triangular matrix T.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:

```
T*x = w*x,     y'*T = w*y'
```

where y' denotes the conjugate transpose of the vector y.

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of T, or the products Q*X and/or Q*Y, where Q is an input unitary

matrix. If T was obtained from the Schur factorization of an original matrix A = Q*T*Q', then Q*X and Q*Y are the matrices of right or left eigenvectors of A.

ARGUMENTS

- **SIDE (input)**

```
= 'R':  compute right eigenvectors only;

= 'L':  compute left eigenvectors only;

= 'B':  compute both right and left eigenvectors.
```

- **HOWMNY (input)**

```
= 'A':  compute all right and/or left eigenvectors;

 = 'B':  compute all right and/or left eigenvectors,
and backtransform them using the input matrices
supplied in VR and/or VL;
 = 'S':  compute selected right and/or left eigenvectors,
```

specified by the logical array SELECT.

- **SELECT (input/output)**
  If HOWMNY = 'S', SELECT specifies the eigenvectors to be computed. If HOWMNY = 'A' or 'B', SELECT is not referenced. To select the eigenvector corresponding to the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**
  The order of the matrix T. N >= 0.
- **T (input/output)**
  The upper triangular matrix T. T is modified, but restored on exit.
- **LDT (input)**
  The leading dimension of the array T. LDT >= max(1,N).
- **VL (input/output)**
  On entry, if SIDE = 'L' or 'B' and HOWMNY = 'B', VL must contain an N-by-N matrix Q (usually the unitary matrix Q of Schur vectors returned by CHSEQR). On exit, if SIDE = 'L' or 'B', VL contains: if HOWMNY = 'A', the matrix Y of left eigenvectors of T; VL is lower triangular. The i-th column `VL(i)` of VL is the eigenvector corresponding to T(i,i). if HOWMNY = 'B', the matrix Q*Y; if HOWMNY = 'S', the left eigenvectors of T specified by SELECT, stored consecutively in the columns of VL, in the same order as their eigenvalues. If SIDE = 'R', VL is not referenced.
- **LDVL (input)**
  The leading dimension of the array VL. LDVL >= `max(1,N)` if SIDE = 'L' or 'B'; LDVL >= 1 otherwise.
- **VR (input/output)**
  On entry, if SIDE = 'R' or 'B' and HOWMNY = 'B', VR must contain an N-by-N matrix Q (usually the unitary matrix Q of Schur vectors returned by CHSEQR). On exit, if SIDE = 'R' or 'B', VR contains: if HOWMNY = 'A', the matrix X of right eigenvectors of T; VR is upper triangular. The i-th column `VR(i)` of VR is the eigenvector corresponding to T(i,i). if HOWMNY = 'B', the matrix Q*X; if HOWMNY = 'S', the right eigenvectors of T specified by SELECT, stored consecutively in the columns of VR, in the same order as their eigenvalues. If SIDE = 'L', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR >= `max(1,N)` if SIDE = 'R' or 'B'; LDVR >= 1 otherwise.
- **MM (input)**
  The number of columns in the arrays VL and/or VR. MM >= M.
- **M (output)**
  The number of columns in the arrays VL and/or VR actually used to store the eigenvectors. If HOWMNY = 'A' or 'B', M is set to N. Each selected eigenvector occupies one column.
- **WORK (workspace)**
  `dimension(2*N)`
- **RWORK (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The algorithm used in this program is basically backward (forward) substitution, with scaling to make the the code robust against possible overflow.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x| + |y|.

NAME

ztrexc - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST

SYNOPSIS

```
SUBROUTINE ZTREXC( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, INFO)
CHARACTER * 1 COMPQ
DOUBLE COMPLEX T(LDT,*), Q(LDQ,*)
INTEGER N, LDT, LDQ, IFST, ILST, INFO

SUBROUTINE ZTREXC_64( COMPQ, N, T, LDT, Q, LDQ, IFST, ILST, INFO)
CHARACTER * 1 COMPQ
DOUBLE COMPLEX T(LDT,*), Q(LDQ,*)
INTEGER*8 N, LDT, LDQ, IFST, ILST, INFO
```

F95 INTERFACE

```
SUBROUTINE TREXC( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST, [INFO])
CHARACTER(LEN=1) :: COMPQ
COMPLEX(8), DIMENSION(:,:) :: T, Q
INTEGER :: N, LDT, LDQ, IFST, ILST, INFO

SUBROUTINE TREXC_64( COMPQ, [N], T, [LDT], Q, [LDQ], IFST, ILST,
*       [INFO])
CHARACTER(LEN=1) :: COMPQ
COMPLEX(8), DIMENSION(:,:) :: T, Q
INTEGER(8) :: N, LDT, LDQ, IFST, ILST, INFO
```

C INTERFACE

#include <sunperf.h>

void ztrexc(char compq, int n, doublecomplex *t, int ldt, doublecomplex *q, int ldq, int ifst, int ilst, int *info);

void ztrexc_64(char compq, long n, doublecomplex *t, long ldt, doublecomplex *q, long ldq, long ifst, long ilst, long *info);

PURPOSE

ztrexc reorders the Schur factorization of a complex matrix A = Q*T*Q**H, so that the diagonal element of T with row index IFST is moved to row ILST.

The Schur form T is reordered by a unitary similarity transformation Z**H*T*Z, and optionally the matrix Q of Schur vectors is updated by postmultplying it with Z.

ARGUMENTS

- **COMPQ (input)**

```
= 'V':  update the matrix Q of Schur vectors;

= 'N':  do not update Q.
```

- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input/output)**
  On entry, the upper triangular matrix T. On exit, the reordered upper triangular matrix.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the unitary transformation matrix Z which reorders T. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **IFST (input)**
  Specify the reordering of the diagonal elements of T: The element with row index IFST is moved to row ILST by a sequence of transpositions between adjacent elements. 1 < = IFST < = N; 1 < = ILST < = N.
- **ILST (input)**
  See the description of IFST.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztrmm - perform one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

SYNOPSIS

```
 SUBROUTINE ZTRMM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER M, N, LDA, LDB

 SUBROUTINE ZTRMM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 M, N, LDA, LDB
```

F95 INTERFACE

```
 SUBROUTINE TRMM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, LDA, LDB

 SUBROUTINE TRMM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, LDA, LDB
```

C INTERFACE

#include <sunperf.h>

void ztrmm(char side, char uplo, char transa, char diag, int m, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb);

void ztrmm_64(char side, char uplo, char transa, char diag, long m, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb);

PURPOSE

ztrmm performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A' or op(A) = conjg(A')

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) multiplies B from the left or right as follows:

  SIDE = 'L' or 'l' B : = alpha*op(A)*B.

  SIDE = 'R' or 'r' B : = alpha*B*op(A).

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = conjg(A').

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity.

  Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1,M), when SIDE = 'R' or 'r' then LDA > = max(1,N). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading M by N part of the array B must contain the matrix B, and on exit is overwritten by the transformed matrix.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling subprogram. LDB must be at least max(1,M). Unchanged on exit.

NAME

ztrmv - perform one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x

---

SYNOPSIS

```
SUBROUTINE ZTRMV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER N, LDA, INCY

SUBROUTINE ZTRMV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TRMV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCY

SUBROUTINE TRMV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ztrmv(char uplo, char transa, char diag, int n, doublecomplex *a, int lda, doublecomplex *y, int incy);

void ztrmv_64(char uplo, char transa, char diag, long n, doublecomplex *a, long lda, doublecomplex *y, long incy);

PURPOSE

ztrmv performs one of the matrix-vector operations x := A*x, or x := A'*x, or x := conjg(A')*x where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the operation to be performed as follows:

  TRANSA = 'N' or 'n' x : = A*x.

  TRANSA = 'T' or 't' x : = A'*x.

  TRANSA = 'C' or 'c' x : = conjg(A')*x.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element vector x. On exit, Y is overwritten with the tranformed vector x.
- **INCY (input)**
  On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztrrfs - provide error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix

---

SYNOPSIS

```
 SUBROUTINE ZTRRFS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB, X,
*      LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER N, NRHS, LDA, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)

 SUBROUTINE ZTRRFS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      X, LDX, FERR, BERR, WORK, WORK2, INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), X(LDX,*), WORK(*)
 INTEGER*8 N, NRHS, LDA, LDB, LDX, INFO
 DOUBLE PRECISION FERR(*), BERR(*), WORK2(*)
```

F95 INTERFACE

```
 SUBROUTINE TRRFS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, X
 INTEGER :: N, NRHS, LDA, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2

 SUBROUTINE TRRFS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], X, [LDX], FERR, BERR, [WORK], [WORK2], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:) :: WORK
 COMPLEX(8), DIMENSION(:,:) :: A, B, X
 INTEGER(8) :: N, NRHS, LDA, LDB, LDX, INFO
 REAL(8), DIMENSION(:) :: FERR, BERR, WORK2
```

C INTERFACE

#include <sunperf.h>

void ztrrfs(char uplo, char transa, char diag, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *x, int ldx, double *ferr, double *berr, int *info);

void ztrrfs_64(char uplo, char transa, char diag, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *x, long ldx, double *ferr, double *berr, long *info);

PURPOSE

ztrrfs provides error bounds and backward error estimates for the solution to a system of linear equations with a triangular coefficient matrix.

The solution matrix X must be computed by CTRTRS or some other means before entering this routine. CTRRFS does not do iterative refinement because doing so cannot improve the backward error.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is

not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,N).
- **B (input)**
  The right hand side matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **X (input)**
  The solution matrix X.
- **LDX (input)**
  The leading dimension of the array X. LDX >= max(1,N).
- **FERR (output)**
  The estimated forward error bound for each solution vector `X(j)` (the j-th column of the solution matrix X). If XTRUE is the true solution corresponding to X(j), `FERR(j)` is an estimated upper bound for the magnitude of the largest element in (X(j) - XTRUE) divided by the magnitude of the largest element in X(j). The estimate is as reliable as the estimate for RCOND, and is almost always a slight overestimate of the true error.
- **BERR (output)**
  The componentwise relative backward error of each solution vector `X(j)` (i.e., the smallest relative change in any element of A or B that makes `X(j)` an exact solution).
- **WORK (workspace)**
  `dimension(2*N)`
- **WORK2 (workspace)**
  `dimension(N)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

ztrsen - reorder the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace

---

SYNOPSIS

```
 SUBROUTINE ZTRSEN( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, W, M, S,
*       SEP, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 DOUBLE COMPLEX T(LDT,*), Q(LDQ,*), W(*), WORK(*)
 INTEGER N, LDT, LDQ, M, LWORK, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION S, SEP

 SUBROUTINE ZTRSEN_64( JOB, COMPQ, SELECT, N, T, LDT, Q, LDQ, W, M,
*       S, SEP, WORK, LWORK, INFO)
 CHARACTER * 1 JOB, COMPQ
 DOUBLE COMPLEX T(LDT,*), Q(LDQ,*), W(*), WORK(*)
 INTEGER*8 N, LDT, LDQ, M, LWORK, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION S, SEP
```

F95 INTERFACE

```
 SUBROUTINE TRSEN( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], W, M,
*       S, SEP, WORK, [LWORK], [INFO])
 CHARACTER(LEN=1) :: JOB, COMPQ
 COMPLEX(8), DIMENSION(:) :: W, WORK
 COMPLEX(8), DIMENSION(:,:) :: T, Q
 INTEGER :: N, LDT, LDQ, M, LWORK, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8) :: S, SEP

 SUBROUTINE TRSEN_64( JOB, COMPQ, SELECT, [N], T, [LDT], Q, [LDQ], W,
*       M, S, SEP, WORK, [LWORK], [INFO])
```

```
CHARACTER(LEN=1) :: JOB, COMPQ
COMPLEX(8), DIMENSION(:) :: W, WORK
COMPLEX(8), DIMENSION(:,:) :: T, Q
INTEGER(8) :: N, LDT, LDQ, M, LWORK, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8) :: S, SEP
```

C INTERFACE

#include <sunperf.h>

void ztrsen(char job, char compq, logical *select, int n, doublecomplex *t, int ldt, doublecomplex *q, int ldq, doublecomplex *w, int *m, double *s, double *sep, doublecomplex *work, int lwork, int *info);

void ztrsen_64(char job, char compq, logical *select, long n, doublecomplex *t, long ldt, doublecomplex *q, long ldq, doublecomplex *w, long *m, double *s, double *sep, doublecomplex *work, long lwork, long *info);

PURPOSE

ztrsen reorders the Schur factorization of a complex matrix A = Q*T*Q**H, so that a selected cluster of eigenvalues appears in the leading positions on the diagonal of the upper triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for the cluster of eigenvalues (S) or the invariant subspace (SEP):

  ```
  = 'N': none;

  = 'E': for eigenvalues only (S);

  = 'V': for invariant subspace only (SEP);

  = 'B': for both eigenvalues and invariant subspace (S and
  SEP).
  ```
- **COMPQ (input)**

  ```
  = 'V': update the matrix Q of Schur vectors;

  = 'N': do not update Q.
  ```
- **SELECT (input)**
  SELECT specifies the eigenvalues in the selected cluster. To select the j-th eigenvalue, `SELECT(j)` must be set to .TRUE..
- **N (input)**

The order of the matrix T. N > = 0.

- **T (input/output)**
  On entry, the upper triangular matrix T. On exit, T is overwritten by the reordered matrix T, with the selected eigenvalues as the leading diagonal elements.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **Q (input/output)**
  On entry, if COMPQ = 'V', the matrix Q of Schur vectors. On exit, if COMPQ = 'V', Q has been postmultiplied by the unitary transformation matrix which reorders T; the leading M columns of Q form an orthonormal basis for the specified invariant subspace. If COMPQ = 'N', Q is not referenced.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = 1; and if COMPQ = 'V', LDQ > = N.
- **W (output)**
  The reordered eigenvalues of T, in the same order as they appear on the diagonal of T.
- **M (output)**
  The dimension of the specified invariant subspace. 0 < = M < = N.
- **S (output)**
  If JOB = 'E' or 'B', S is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. S cannot underestimate the true reciprocal condition number by more than a factor of sqrt(N). If M = 0 or N, S = 1. If JOB = 'N' or 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', SEP is the estimated reciprocal condition number of the specified invariant subspace. If M = 0 or N, SEP = norm(T). If JOB = 'N' or 'E', SEP is not referenced.
- **WORK (output)**
  If JOB = 'N', WORK is not referenced. Otherwise, on exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If JOB = 'N', LWORK > = 1; if JOB = 'E', LWORK = M*(N-M); if JOB = 'V' or 'B', LWORK > = 2*M*(N-M).

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

CTRSEN first collects the selected eigenvalues by computing a unitary transformation Z to move them to the top left corner of T. In other words, the selected eigenvalues are the eigenvalues of T11 in:

```
Z'*T*Z  = ( T11 T12 ) n1

          (  0  T22 ) n2

             n1  n2
```

where N = n1+n2 and Z' means the conjugate transpose of Z. The first n1 columns of Z span the specified invariant subspace

of T.

If T has been obtained from the Schur factorization of a matrix A = Q*T*Q', then the reordered Schur factorization of A is given by A = (Q*Z)*(Z'*T*Z)*(Q*Z)', and the first n1 columns of Q*Z span the corresponding invariant subspace of A.

The reciprocal condition number of the average of the eigenvalues of T11 may be returned in S. S lies between 0 (very badly conditioned) and 1 (very well conditioned). It is computed as follows. First we compute R so that

```
P  = ( I  R ) n1

     ( 0  0 ) n2

       n1 n2
```

is the projector on the invariant subspace associated with T11. R is the solution of the Sylvester equation:

```
T11*R - R*T22  = T12.
```

Let F-norm(M) denote the Frobenius-norm of M and 2-norm(M) denote the two-norm of M. Then S is computed as the lower bound

```
(1 + F-norm(R)**2)**(-1/2)
```

on the reciprocal of 2-norm(P), the true reciprocal condition number. S cannot underestimate 1 / 2-norm(P) by more than a factor of sqrt(N).

An approximate error bound for the computed average of the eigenvalues of T11 is

```
EPS * norm(T) / S
```

where EPS is the machine precision.

The reciprocal condition number of the right invariant subspace spanned by the first n1 columns of Z (or of Q*Z) is returned in SEP. SEP is defined as the separation of T11 and T22:

```
sep( T11, T22 )  = sigma-min( C )
```

where sigma-min(C) is the smallest singular value of the

n1*n2-by-n1*n2 matrix

```
C   = kprod( I(n2), T11 ) - kprod( transpose(T22), I(n1) )
```

`I(m)` is an m by m identity matrix, and kprod denotes the Kronecker product. We estimate sigma-min(C) by the reciprocal of an estimate of the 1-norm of inverse(C). The true reciprocal 1-norm of `inverse(C)` cannot differ from sigma-min(C) by more than a factor of sqrt(n1*n2).

When SEP is small, small changes in T can cause large changes in the invariant subspace. An approximate bound on the maximum angular error in the computed right invariant subspace is

```
EPS * norm(T) / SEP
```

NAME

ztrsm - solve one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B

SYNOPSIS

```
 SUBROUTINE ZTRSM( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B,
*      LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER M, N, LDA, LDB

 SUBROUTINE ZTRSM_64( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA,
*      B, LDB)
 CHARACTER * 1 SIDE, UPLO, TRANSA, DIAG
 DOUBLE COMPLEX ALPHA
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 M, N, LDA, LDB
```

F95 INTERFACE

```
 SUBROUTINE TRSM( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: M, N, LDA, LDB

 SUBROUTINE TRSM_64( SIDE, UPLO, [TRANSA], DIAG, [M], [N], ALPHA, A,
*       [LDA], B, [LDB])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANSA, DIAG
 COMPLEX(8) :: ALPHA
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: M, N, LDA, LDB
```

C INTERFACE

#include <sunperf.h>

void ztrsm(char side, char uplo, char transa, char diag, int m, int n, doublecomplex alpha, doublecomplex *a, int lda, doublecomplex *b, int ldb);

void ztrsm_64(char side, char uplo, char transa, char diag, long m, long n, doublecomplex alpha, doublecomplex *a, long lda, doublecomplex *b, long ldb);

PURPOSE

ztrsm solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of

```
op( A ) = A   or   op( A ) = A'   or   op( A ) = conjg( A' ).
```

The matrix X is overwritten on B.

ARGUMENTS

- **SIDE (input)**
  On entry, SIDE specifies whether op(A) appears on the left or right of X as follows:

  SIDE = 'L' or 'l' op(A)*X = alpha*B.

  SIDE = 'R' or 'r' X*op(A) = alpha*B.

  Unchanged on exit.
- **UPLO (input)**
  On entry, UPLO specifies whether the matrix A is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the form of op(A) to be used in the matrix multiplication as follows:

  TRANSA = 'N' or 'n' op(A) = A.

  TRANSA = 'T' or 't' op(A) = A'.

  TRANSA = 'C' or 'c' op(A) = conjg(A').

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

DIAG = 'U' or 'u' A is assumed to be unit triangular.

DIAG = 'N' or 'n' A is not assumed to be unit triangular.

Unchanged on exit.

- **M (input)**
  On entry, M specifies the number of rows of B. M > = 0. Unchanged on exit.
- **N (input)**
  On entry, N specifies the number of columns of B. N > = 0. Unchanged on exit.
- **ALPHA (input)**
  On entry, ALPHA specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry. Unchanged on exit.
- **A (input)**
  when SIDE = 'L' or 'l' and is n when SIDE = 'R' or 'r'.

  Before entry with UPLO = 'U' or 'u', the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.

  Before entry with UPLO = 'L' or 'l', the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.

  Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity.

  Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When SIDE = 'L' or 'l' then LDA > = max(1,M), when SIDE = 'R' or 'r' then LDA > = max(1,N). Unchanged on exit.
- **B (input/output)**
  Before entry, the leading M by N part of the array B must contain the right-hand side matrix B, and on exit is overwritten by the solution matrix X.
- **LDB (input)**
  On entry, LDB specifies the first dimension of B as declared in the calling subprogram. LDB > = max(1,M). Unchanged on exit.

NAME

ztrsna - estimate reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary)

SYNOPSIS

```
 SUBROUTINE ZTRSNA( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 DOUBLE COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(LDWORK,*)
 INTEGER N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL SELECT(*)
 DOUBLE PRECISION S(*), SEP(*), WORK1(*)

 SUBROUTINE ZTRSNA_64( JOB, HOWMNY, SELECT, N, T, LDT, VL, LDVL, VR,
*      LDVR, S, SEP, MM, M, WORK, LDWORK, WORK1, INFO)
 CHARACTER * 1 JOB, HOWMNY
 DOUBLE COMPLEX T(LDT,*), VL(LDVL,*), VR(LDVR,*), WORK(LDWORK,*)
 INTEGER*8 N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL*8 SELECT(*)
 DOUBLE PRECISION S(*), SEP(*), WORK1(*)
```

F95 INTERFACE

```
 SUBROUTINE TRSNA( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 COMPLEX(8), DIMENSION(:,:) :: T, VL, VR, WORK
 INTEGER :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
 LOGICAL, DIMENSION(:) :: SELECT
 REAL(8), DIMENSION(:) :: S, SEP, WORK1

 SUBROUTINE TRSNA_64( JOB, HOWMNY, SELECT, [N], T, [LDT], VL, [LDVL],
*       VR, [LDVR], S, SEP, MM, M, [WORK], [LDWORK], [WORK1], [INFO])
 CHARACTER(LEN=1) :: JOB, HOWMNY
 COMPLEX(8), DIMENSION(:,:) :: T, VL, VR, WORK
```

```
INTEGER(8) :: N, LDT, LDVL, LDVR, MM, M, LDWORK, INFO
LOGICAL(8), DIMENSION(:) :: SELECT
REAL(8), DIMENSION(:) :: S, SEP, WORK1
```

C INTERFACE

#include <sunperf.h>

void ztrsna(char job, char howmny, logical *select, int n, doublecomplex *t, int ldt, doublecomplex *vl, int ldvl, doublecomplex *vr, int ldvr, double *s, double *sep, int mm, int *m, int ldwork, int *info);

void ztrsna_64(char job, char howmny, logical *select, long n, doublecomplex *t, long ldt, doublecomplex *vl, long ldvl, doublecomplex *vr, long ldvr, double *s, double *sep, long mm, long *m, long ldwork, long *info);

PURPOSE

ztrsna estimates reciprocal condition numbers for specified eigenvalues and/or right eigenvectors of a complex upper triangular matrix T (or of any matrix Q*T*Q**H with Q unitary).

ARGUMENTS

- **JOB (input)**
  Specifies whether condition numbers are required for eigenvalues (S) or eigenvectors (SEP):

  ```
  = 'E': for eigenvalues only (S);

  = 'V': for eigenvectors only (SEP);

  = 'B': for both eigenvalues and eigenvectors (S and SEP).
  ```
- **HOWMNY (input)**

  ```
  = 'A': compute condition numbers for all eigenpairs;

  = 'S': compute condition numbers for selected eigenpairs
  specified by the array SELECT.
  ```
- **SELECT (input)**
  If HOWMNY = 'S', SELECT specifies the eigenpairs for which condition numbers are required. To select condition numbers for the j-th eigenpair, `SELECT(j)` must be set to .TRUE.. If HOWMNY = 'A', SELECT is not referenced.
- **N (input)**
  The order of the matrix T. N > = 0.
- **T (input)**
  The upper triangular matrix T.
- **LDT (input)**
  The leading dimension of the array T. LDT > = max(1,N).
- **VL (input)**
  If JOB = 'E' or 'B', VL must contain left eigenvectors of T (or of any Q*T*Q**H with Q unitary), corresponding to

the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VL, as returned by CHSEIN or CTREVC. If JOB = 'V', VL is not referenced.

- **LDVL (input)**
  The leading dimension of the array VL. LDVL > = 1; and if JOB = 'E' or 'B', LDVL > = N.
- **VR (input)**
  If JOB = 'E' or 'B', VR must contain right eigenvectors of T (or of any Q*T*Q**H with Q unitary), corresponding to the eigenpairs specified by HOWMNY and SELECT. The eigenvectors must be stored in consecutive columns of VR, as returned by CHSEIN or CTREVC. If JOB = 'V', VR is not referenced.
- **LDVR (input)**
  The leading dimension of the array VR. LDVR > = 1; and if JOB = 'E' or 'B', LDVR > = N.
- **S (output)**
  If JOB = 'E' or 'B', the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. Thus S(j), SEP(j), and the j-th columns of VL and VR all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected). If JOB = 'V', S is not referenced.
- **SEP (output)**
  If JOB = 'V' or 'B', the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If JOB = 'E', SEP is not referenced.
- **MM (input)**
  The number of elements in the arrays S (if JOB = 'E' or 'B') and/or SEP (if JOB = 'V' or 'B'). MM > = M.
- **M (output)**
  The number of elements of the arrays S and/or SEP actually used to store the estimated condition numbers. If HOWMNY = 'A', M is set to N.
- **WORK (workspace)**
  `dimension(LDWORK,N+1)` If JOB = 'E', WORK is not referenced.
- **LDWORK (input)**
  The leading dimension of the array WORK. LDWORK > = 1; and if JOB = 'V' or 'B', LDWORK > = N.
- **WORK1 (workspace)**
  `dimension(N)` If JOB = 'E', WORK1 is not referenced.
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

The reciprocal of the condition number of an eigenvalue lambda is defined as

```
S(lambda)  = |v'*u| / (norm(u)*norm(v))
```

where u and v are the right and left eigenvectors of T corresponding to lambda; v' denotes the conjugate transpose of v, and `norm(u)` denotes the Euclidean norm. These reciprocal condition numbers always lie between zero (very badly conditioned) and one (very well conditioned). If n = 1, `S(lambda)` is defined to be 1.

An approximate error bound for a computed eigenvalue `W(i)` is given by

```
EPS * norm(T) / S(i)
```

where EPS is the machine precision.

The reciprocal of the condition number of the right eigenvector u corresponding to lambda is defined as follows. Suppose

```
T  = ( lambda   c   )

     (   0     T22 )
```

Then the reciprocal condition number is

```
SEP( lambda, T22 )  = sigma-min( T22 - lambda*I )
```

where sigma-min denotes the smallest singular value. We approximate the smallest singular value by the reciprocal of an estimate of the one-norm of the inverse of T22 - lambda*I. If n = 1, SEP(1) is defined to be abs(T(1,1)).

An approximate error bound for a computed right eigenvector VR(i) is given by

```
EPS * norm(T) / SEP(i)
```

NAME

ztrsv - solve one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b

---

SYNOPSIS

```
SUBROUTINE ZTRSV( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER N, LDA, INCY

SUBROUTINE ZTRSV_64( UPLO, TRANSA, DIAG, N, A, LDA, Y, INCY)
CHARACTER * 1 UPLO, TRANSA, DIAG
DOUBLE COMPLEX A(LDA,*), Y(*)
INTEGER*8 N, LDA, INCY
```

F95 INTERFACE

```
SUBROUTINE TRSV( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INCY

SUBROUTINE TRSV_64( UPLO, [TRANSA], DIAG, [N], A, [LDA], Y, [INCY])
CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
COMPLEX(8), DIMENSION(:) :: Y
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INCY
```

C INTERFACE

#include <sunperf.h>

void ztrsv(char uplo, char transa, char diag, int n, doublecomplex *a, int lda, doublecomplex *y, int incy);

void ztrsv_64(char uplo, char transa, char diag, long n, doublecomplex *a, long lda, doublecomplex *y, long incy);

PURPOSE

ztrsv solves one of the systems of equations A*x = b, or A'*x = b, or conjg(A')*x = b where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

ARGUMENTS

- **UPLO (input)**
  On entry, UPLO specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u' A is an upper triangular matrix.

  UPLO = 'L' or 'l' A is a lower triangular matrix.

  Unchanged on exit.
- **TRANSA (input)**
  On entry, TRANSA specifies the equations to be solved as follows:

  TRANSA = 'N' or 'n' A*x = b.

  TRANSA = 'T' or 't' A'*x = b.

  TRANSA = 'C' or 'c' conjg(A')*x = b.

  Unchanged on exit.
- **DIAG (input)**
  On entry, DIAG specifies whether or not A is unit triangular as follows:

  DIAG = 'U' or 'u' A is assumed to be unit triangular.

  DIAG = 'N' or 'n' A is not assumed to be unit triangular.

  Unchanged on exit.
- **N (input)**
  On entry, N specifies the order of the matrix A. N > = 0. Unchanged on exit.
- **A (input)**
  Before entry with UPLO = 'U' or 'u', the leading n by n upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Before entry with UPLO = 'L' or 'l', the leading n by n lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced. Note that when DIAG = 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity. Unchanged on exit.
- **LDA (input)**
  On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA > = max(1, n). Unchanged on exit.
- **Y (input/output)**
  (1 + (n - 1)*abs(INCY)). Before entry, the incremented array Y must contain the n element right-hand side vector b. On exit, Y is overwritten with the solution vector x.
- **INCY (input)**

On entry, INCY specifies the increment for the elements of Y. INCY < > 0. Unchanged on exit.

NAME

ztrsyl - solve the complex Sylvester matrix equation

SYNOPSIS

```
 SUBROUTINE ZTRSYL( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C, LDC,
*      SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER ISGN, M, N, LDA, LDB, LDC, INFO
 DOUBLE PRECISION SCALE

 SUBROUTINE ZTRSYL_64( TRANA, TRANB, ISGN, M, N, A, LDA, B, LDB, C,
*      LDC, SCALE, INFO)
 CHARACTER * 1 TRANA, TRANB
 DOUBLE COMPLEX A(LDA,*), B(LDB,*), C(LDC,*)
 INTEGER*8 ISGN, M, N, LDA, LDB, LDC, INFO
 DOUBLE PRECISION SCALE
```

F95 INTERFACE

```
 SUBROUTINE TRSYL( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL(8) :: SCALE

 SUBROUTINE TRSYL_64( TRANA, TRANB, ISGN, [M], [N], A, [LDA], B, [LDB],
*       C, [LDC], SCALE, [INFO])
 CHARACTER(LEN=1) :: TRANA, TRANB
 COMPLEX(8), DIMENSION(:,:) :: A, B, C
 INTEGER(8) :: ISGN, M, N, LDA, LDB, LDC, INFO
 REAL(8) :: SCALE
```

C INTERFACE

#include <sunperf.h>

void ztrsyl(char trana, char tranb, int isgn, int m, int n, doublecomplex *a, int lda, doublecomplex *b, int ldb, doublecomplex *c, int ldc, double *scale, int *info);

void ztrsyl_64(char trana, char tranb, long isgn, long m, long n, doublecomplex *a, long lda, doublecomplex *b, long ldb, doublecomplex *c, long ldc, double *scale, long *info);

PURPOSE

ztrsyl solves the complex Sylvester matrix equation:

```
op(A)*X + X*op(B) = scale*C or

op(A)*X - X*op(B) = scale*C,
```

where `op(A)` = A or A**H, and A and B are both upper triangular. A is M-by-M and B is N-by-N; the right hand side C and the solution X are M-by-N; and scale is an output scale factor, set <= 1 to avoid overflow in X.

ARGUMENTS

- **TRANA (input)**
  Specifies the option op(A):

  ```
  = 'N': op(A)  = A    (No transpose)

  = 'C': op(A)  = A**H (Conjugate transpose)
  ```
- **TRANB (input)**
  Specifies the option op(B):

  ```
  = 'N': op(B)  = B    (No transpose)

  = 'C': op(B)  = B**H (Conjugate transpose)
  ```
- **ISGN (input)**
  Specifies the sign in the equation:

  ```
  = +1: solve op(A)*X + X*op(B)  = scale*C

  = -1: solve op(A)*X - X*op(B)  = scale*C
  ```
- **M (input)**
  The order of the matrix A, and the number of rows in the matrices X and C. M > = 0.
- **N (input)**
  The order of the matrix B, and the number of columns in the matrices X and C. N > = 0.
- **A (input)**

The upper triangular matrix A.

- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **B (input)**
  The upper triangular matrix B.
- **LDB (input)**
  The leading dimension of the array B. LDB >= max(1,N).
- **C (input/output)**
  On entry, the M-by-N right hand side matrix C. On exit, C is overwritten by the solution matrix X.
- **LDC (input)**
  The leading dimension of the array C. LDC >= `max(1,M)`
- **SCALE (output)**
  The scale factor, scale, set <= 1 to avoid overflow in X.
- **INFO (output)**

```
 = 0: successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 = 1: A and B have common or very close eigenvalues; perturbed
values were used to solve the equation (but the matrices
A and B are unchanged).
```

NAME

ztrti2 - compute the inverse of a complex upper or lower triangular matrix

---

SYNOPSIS

```
SUBROUTINE ZTRTI2( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE ZTRTI2_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TRTI2( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE TRTI2_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ztrti2(char uplo, char diag, int n, doublecomplex *a, int lda, int *info);

void ztrti2_64(char uplo, char diag, long n, doublecomplex *a, long lda, long *info);

PURPOSE

ztrti2 computes the inverse of a complex upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**
  Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular

  ```
  = 'L':  Lower triangular
  ```
- **DIAG (input)**
  Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular

  ```
  = 'U':  Unit triangular
  ```
- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading n by n upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading n by n lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

  On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -k, the k-th argument had an illegal value
  ```

NAME

ztrtri - compute the inverse of a complex upper or lower triangular matrix A

SYNOPSIS

```
SUBROUTINE ZTRTRI( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(LDA,*)
INTEGER N, LDA, INFO

SUBROUTINE ZTRTRI_64( UPLO, DIAG, N, A, LDA, INFO)
CHARACTER * 1 UPLO, DIAG
DOUBLE COMPLEX A(LDA,*)
INTEGER*8 N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TRTRI( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, INFO

SUBROUTINE TRTRI_64( UPLO, DIAG, [N], A, [LDA], [INFO])
CHARACTER(LEN=1) :: UPLO, DIAG
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ztrtri(char uplo, char diag, int n, doublecomplex *a, int lda, int *info);

void ztrtri_64(char uplo, char diag, long n, doublecomplex *a, long lda, long *info);

PURPOSE

ztrtri computes the inverse of a complex upper or lower triangular matrix A.

This is the Level 3 BLAS version of the algorithm.

ARGUMENTS

- **UPLO (input)**

```
= 'U':  A is upper triangular;

= 'L':  A is lower triangular.
```

- **DIAG (input)**

```
= 'N':  A is non-unit triangular;

= 'U':  A is unit triangular.
```

- **N (input)**
  The order of the matrix A. N > = 0.
- **A (input/output)**
  On entry, the triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1. On exit, the (triangular) inverse of the original matrix, in the same storage format.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value

> 0: if INFO  = i, A(i,i) is exactly zero.  The triangular
matrix is singular and its inverse can not be computed.
```

NAME

ztrtrs - solve a triangular system of the form A * X = B, A**T * X = B, or A**H * X = B,

---

SYNOPSIS

```
 SUBROUTINE ZTRTRS( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER N, NRHS, LDA, LDB, INFO

 SUBROUTINE ZTRTRS_64( UPLO, TRANSA, DIAG, N, NRHS, A, LDA, B, LDB,
*      INFO)
 CHARACTER * 1 UPLO, TRANSA, DIAG
 DOUBLE COMPLEX A(LDA,*), B(LDB,*)
 INTEGER*8 N, NRHS, LDA, LDB, INFO
```

F95 INTERFACE

```
 SUBROUTINE TRTRS( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER :: N, NRHS, LDA, LDB, INFO

 SUBROUTINE TRTRS_64( UPLO, [TRANSA], DIAG, [N], [NRHS], A, [LDA], B,
*       [LDB], [INFO])
 CHARACTER(LEN=1) :: UPLO, TRANSA, DIAG
 COMPLEX(8), DIMENSION(:,:) :: A, B
 INTEGER(8) :: N, NRHS, LDA, LDB, INFO
```

C INTERFACE

#include <sunperf.h>

void ztrtrs(char uplo, char transa, char diag, int n, int nrhs, doublecomplex *a, int lda, doublecomplex *b, int ldb, int *info);

void ztrtrs_64(char uplo, char transa, char diag, long n, long nrhs, doublecomplex *a, long lda, doublecomplex *b, long ldb, long *info);

PURPOSE

ztrtrs solves a triangular system of the form

where A is a triangular matrix of order N, and B is an N-by-NRHS matrix. A check is made to verify that A is nonsingular.

ARGUMENTS

- **UPLO (input)**

  ```
  = 'U':  A is upper triangular;

  = 'L':  A is lower triangular.
  ```

- **TRANSA (input)**
  Specifies the form of the system of equations:

  ```
  = 'N':  A * X  = B     (No transpose)

  = 'T':  A**T * X  = B  (Transpose)

  = 'C':  A**H * X  = B  (Conjugate transpose)
  ```

- **DIAG (input)**

  ```
  = 'N':  A is non-unit triangular;

  = 'U':  A is unit triangular.
  ```

- **N (input)**
  The order of the matrix A. N > = 0.

- **NRHS (input)**
  The number of right hand sides, i.e., the number of columns of the matrix B. NRHS > = 0.

- **A (input)**
  The triangular matrix A. If UPLO = 'U', the leading N-by-N upper triangular part of the array A contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If UPLO = 'L', the leading N-by-N lower triangular part of the array A contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced. If DIAG = 'U', the diagonal elements of A are also not referenced and are assumed to be 1.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).

- **B (input/output)**
  On entry, the right hand side matrix B. On exit, if INFO = 0, the solution matrix X.
- **LDB (input)**
  The leading dimension of the array B. LDB > = max(1,N).
- **INFO (output)**

```
 = 0:  successful exit

 < 0: if INFO  = -i, the i-th argument had an illegal value

 > 0: if INFO  = i, the i-th diagonal element of A is zero,
indicating that the matrix is singular and the solutions
X have not been computed.
```

NAME

ztzrqf - routine is deprecated and has been replaced by routine CTZRZF

SYNOPSIS

```
SUBROUTINE ZTZRQF( M, N, A, LDA, TAU, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*)
INTEGER M, N, LDA, INFO

SUBROUTINE ZTZRQF_64( M, N, A, LDA, TAU, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*)
INTEGER*8 M, N, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE TZRQF( [M], [N], A, [LDA], TAU, [INFO])
COMPLEX(8), DIMENSION(:) :: TAU
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, INFO

SUBROUTINE TZRQF_64( [M], [N], A, [LDA], TAU, [INFO])
COMPLEX(8), DIMENSION(:) :: TAU
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void ztzrqf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void ztzrqf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

ztzrqf routine is deprecated and has been replaced by routine CTZRZF.

CTZRQF reduces the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N unitary matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= M.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the unitary matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), whose conjugate transpose is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
Z( k )  = ( I     0   ),

          ( 0  T( k ) )
```

where

```
T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                              (   0    )
```

```
                                        ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

ztzrzf - reduce the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations

---

SYNOPSIS

```
SUBROUTINE ZTZRZF( M, N, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, LDA, LWORK, INFO

SUBROUTINE ZTZRZF_64( M, N, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE TZRZF( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, LDA, LWORK, INFO

SUBROUTINE TZRZF_64( [M], [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void ztzrzf(int m, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void ztzrzf_64(long m, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

ztzrzf reduces the M-by-N (M<=N) complex upper trapezoidal matrix A to upper triangular form by means of unitary transformations.

The upper trapezoidal matrix A is factored as

```
A = ( R  0 ) * Z,
```

where Z is an N-by-N unitary matrix and R is an M-by-M upper triangular matrix.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix A. M >= 0.
- **N (input)**
  The number of columns of the matrix A. N >= 0.
- **A (input/output)**
  On entry, the leading M-by-N upper trapezoidal part of the array A must contain the matrix to be factorized. On exit, the leading M-by-M upper triangular part of A contains the upper triangular matrix R, and elements M+1 to N of the first M rows of A, with the array TAU, represent the unitary matrix Z as a product of M elementary reflectors.
- **LDA (input)**
  The leading dimension of the array A. LDA >= max(1,M).
- **TAU (output)**
  The scalar factors of the elementary reflectors.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument had an illegal value
  ```

FURTHER DETAILS

Based on contributions by

```
  A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the (m - k + 1)th row of A, is given in the form

```
   Z( k )  = ( I     0   ),

             ( 0  T( k ) )
```

where

```
   T( k )  = I - tau*u( k )*u( k )',   u( k )  = (   1    ),
                                                (   0    )
                                                ( z( k ) )
```

tau is a scalar and z(k) is an (n - m) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of X.

The scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of A, such that the elements of z(k) are in a(k, m + 1), ..., a(k, n). The elements of R are returned in the upper triangular part of A.

Z is given by

```
   Z  =  Z( 1 ) * Z( 2 ) * ... * Z( m ).
```

NAME

zung2l - generate an m by n complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE ZUNG2L( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE ZUNG2L_64( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNG2L( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNG2L_64( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zung2l(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zung2l_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zung2l L generates an m by n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by CGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

zung2r - generate an m by n complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE ZUNG2R( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE ZUNG2R_64( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNG2R( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNG2R_64( M, [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zung2r(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zung2r_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zung2r R generates an m by n complex matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by CGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.
- **WORK (workspace)**
  `dimension(N)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

zungbr - generate one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form

---

SYNOPSIS

```
SUBROUTINE ZUNGBR( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE ZUNGBR_64( VECT, M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 VECT
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGBR( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGBR_64( VECT, M, [N], K, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: VECT
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zungbr(char vect, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungbr_64(char vect, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungbr generates one of the complex unitary matrices Q or P**H determined by CGEBRD when reducing a complex matrix A to bidiagonal form: A = Q * B * P**H. Q and P**H are defined as products of elementary reflectors `H(i)` or `G(i)` respectively.

If VECT = 'Q', A is assumed to have been an M-by-K matrix, and Q is of order M:

if m >= k, Q = `H(1) H(2) . . . H(k)` and CUNGBR returns the first n columns of Q, where m >= n >= k;

if m < k, Q = `H(1) H(2) . . . H(m-1)` and CUNGBR returns Q as an M-by-M matrix.

If VECT = 'P', A is assumed to have been a K-by-N matrix, and P**H is of order N:

if k < n, P**H = `G(k) . . . G(2) G(1)` and CUNGBR returns the first m rows of P**H, where n >= m >= k;

if k >= n, P**H = `G(n-1) . . . G(2) G(1)` and CUNGBR returns P**H as an N-by-N matrix.

ARGUMENTS

- **VECT (input)**
  Specifies whether the matrix Q or the matrix P**H is required, as defined in the transformation applied by CGEBRD:

  ```
  = 'Q':  generate Q;

  = 'P':  generate P**H.
  ```
- **M (input)**
  The number of rows of the matrix Q or P**H to be returned. M > = 0.
- **N (input)**
  The number of columns of the matrix Q or P**H to be returned. N > = 0. If VECT = 'Q', M > = N > = min(M,K); if VECT = 'P', N > = M > = min(N,K).
- **K (input)**
  If VECT = 'Q', the number of columns in the original M-by-K matrix reduced by CGEBRD. If VECT = 'P', the number of rows in the original K-by-N matrix reduced by CGEBRD. K > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by CGEBRD. On exit, the M-by-N matrix Q or P**H.
- **LDA (input)**
  The leading dimension of the array A. LDA > = M.

- **TAU (input)**
  (min(M,K)) if VECT = 'Q' (min(N,K)) if VECT = 'P' `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or G(i), which determines Q or P**H, as returned by CGEBRD in its array argument TAUQ or TAUP.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,min(M,N)). For optimum performance LWORK >= min(M,N)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunghr - generate a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD

SYNOPSIS

```
SUBROUTINE ZUNGHR( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, ILO, IHI, LDA, LWORK, INFO

SUBROUTINE ZUNGHR_64( N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, ILO, IHI, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGHR( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, ILO, IHI, LDA, LWORK, INFO

SUBROUTINE UNGHR_64( [N], ILO, IHI, A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, ILO, IHI, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunghr(int n, int ilo, int ihi, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zunghr_64(long n, long ilo, long ihi, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zunghr generates a complex unitary matrix Q which is defined as the product of IHI-ILO elementary reflectors of order N, as returned by CGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **N (input)**
  The order of the matrix Q. N > = 0.
- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of CGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). 1 < = ILO < = IHI < = N, if N > 0; ILO =1 and IHI =0, if N =0.
- **IHI (input)**
  See the description of IHI.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by CGEHRD. On exit, the N-by-N unitary matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,N).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEHRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = IHI-ILO. For optimum performance LWORK > = (IHI-ILO)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zungl2 - generate an m-by-n complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE ZUNGL2( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE ZUNGL2_64( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGL2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNGL2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zungl2(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungl2_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungl2 generates an m-by-n complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

```
Q  =  H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. On exit, the m by n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  `= 0: successful exit`

  `< 0: if INFO  = -i, the i-th argument has an illegal value`

NAME

zunglq - generate an M-by-N complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE ZUNGLQ( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE ZUNGLQ_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGLQ( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGLQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunglq(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zunglq_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zunglq generates an M-by-N complex matrix Q with orthonormal rows, which is defined as the first M rows of a product of K elementary reflectors of order N

```
Q  =  H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit;

< 0:  if INFO  = -i, the i-th argument has an illegal value
```

NAME

zungql - generate an M-by-N complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE ZUNGQL( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE ZUNGQL_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGQL( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGQL_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zungql(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungql_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungql generates an M-by-N complex matrix Q with orthonormal columns, which is defined as the last N columns of a product of K elementary reflectors of order M

```
Q  =  H(k) . . . H(2) H(1)
```

as returned by CGEQLF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. M >= N >= 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N >= K >= 0.
- **A (input/output)**
  On entry, the (n-k+i)-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,N). For optimum performance LWORK >= N*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument has an illegal value
```

NAME

zungqr - generate an M-by-N complex matrix Q with orthonormal columns,

---

SYNOPSIS

```
SUBROUTINE ZUNGQR( M, N, K, A, LDA, TAU, WORKIN, LWORKIN, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORKIN(*)
INTEGER M, N, K, LDA, LWORKIN, INFO

SUBROUTINE ZUNGQR_64( M, N, K, A, LDA, TAU, WORKIN, LWORKIN, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORKIN(*)
INTEGER*8 M, N, K, LDA, LWORKIN, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGQR( M, [N], [K], A, [LDA], TAU, [WORKIN], [LWORKIN],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORKIN
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORKIN, INFO

SUBROUTINE UNGQR_64( M, [N], [K], A, [LDA], TAU, [WORKIN], [LWORKIN],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORKIN
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORKIN, INFO
```

C INTERFACE

#include <sunperf.h>

void zungqr(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungqr_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungqr generates an M-by-N complex matrix Q with orthonormal columns, which is defined as the first N columns of a product of K elementary reflectors of order M

```
Q  =  H(1) H(2) . . . H(k)
```

as returned by CGEQRF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. M > = N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. N > = K > = 0.
- **A (input/output)**
  On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.
- **WORKIN (workspace)**
  On exit, if INFO = 0, `WORKIN(1)` returns the optimal LWORKIN.
- **LWORKIN (input)**
  The dimension of the array WORKIN. LWORKIN > = max(1,N). For optimum performance LWORKIN > = N*NB, where NB is the optimal blocksize.

  If LWORKIN = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORKIN array, returns this value as the first entry of the WORKIN array, and no error message related to LWORKIN is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

zungr2 - generate an m by n complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE ZUNGR2( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, INFO

SUBROUTINE ZUNGR2_64( M, N, K, A, LDA, TAU, WORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGR2( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, INFO

SUBROUTINE UNGR2_64( [M], [N], [K], A, [LDA], TAU, [WORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, INFO
```

C INTERFACE

#include <sunperf.h>

void zungr2(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungr2_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungr2 generates an m by n complex matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

```
Q  =  H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M > = 0.
- **N (input)**
  The number of columns of the matrix Q. N > = M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M > = K > = 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. On exit, the m-by-n matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA > = max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.
- **WORK (workspace)**
  `dimension(M)`
- **INFO (output)**

  ```
  = 0: successful exit

  < 0: if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

zungrq - generate an M-by-N complex matrix Q with orthonormal rows,

---

SYNOPSIS

```
SUBROUTINE ZUNGRQ( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER M, N, K, LDA, LWORK, INFO

SUBROUTINE ZUNGRQ_64( M, N, K, A, LDA, TAU, WORK, LWORK, INFO)
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 M, N, K, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGRQ( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK], [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: M, N, K, LDA, LWORK, INFO

SUBROUTINE UNGRQ_64( M, [N], [K], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: M, N, K, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zungrq(int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungrq_64(long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungrq generates an M-by-N complex matrix Q with orthonormal rows, which is defined as the last M rows of a product of K elementary reflectors of order N

```
Q  =  H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF.

ARGUMENTS

- **M (input)**
  The number of rows of the matrix Q. M >= 0.
- **N (input)**
  The number of columns of the matrix Q. N >= M.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. M >= K >= 0.
- **A (input/output)**
  On entry, the (m-k+i)-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. On exit, the M-by-N matrix Q.
- **LDA (input)**
  The first dimension of the array A. LDA >= max(1,M).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK >= max(1,M). For optimum performance LWORK >= M*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

  ```
  = 0:  successful exit

  < 0:  if INFO  = -i, the i-th argument has an illegal value
  ```

NAME

zungtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD

SYNOPSIS

```
SUBROUTINE ZUNGTR( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER N, LDA, LWORK, INFO

SUBROUTINE ZUNGTR_64( UPLO, N, A, LDA, TAU, WORK, LWORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX A(LDA,*), TAU(*), WORK(*)
INTEGER*8 N, LDA, LWORK, INFO
```

F95 INTERFACE

```
SUBROUTINE UNGTR( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER :: N, LDA, LWORK, INFO

SUBROUTINE UNGTR_64( UPLO, [N], A, [LDA], TAU, [WORK], [LWORK],
*       [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: A
INTEGER(8) :: N, LDA, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zungtr(char uplo, int n, doublecomplex *a, int lda, doublecomplex *tau, int *info);

void zungtr_64(char uplo, long n, doublecomplex *a, long lda, doublecomplex *tau, long *info);

PURPOSE

zungtr generates a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors of order N, as returned by CHETRD:

if UPLO = 'U', Q = `H(n-1) . . . H(2)` H(1),

if UPLO = 'L', Q = `H(1) H(2) . . .` H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from CHETRD;
 = 'L': Lower triangle of A contains elementary reflectors
from CHETRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **A (input/output)**
  On entry, the vectors which define the elementary reflectors, as returned by CHETRD. On exit, the N-by-N unitary matrix Q.
- **LDA (input)**
  The leading dimension of the array A. LDA > = N.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHETRD.
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. LWORK > = N-1. For optimum performance LWORK > = (N-1)*NB, where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
 = 0:  successful exit
```

```
< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmbr - VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMBR( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMBR_64( VECT, SIDE, TRANS, M, N, K, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 VECT, SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMBR( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMBR_64( VECT, SIDE, [TRANS], [M], [N], K, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: VECT, SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmbr(char vect, char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmbr_64(char vect, char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmbr VECT = 'Q', CUNMBR overwrites the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N': Q * C C * Q TRANS = 'C': Q**H * C C * Q**H

If VECT = 'P', CUNMBR overwrites the general complex M-by-N matrix C with

```
                SIDE = 'L'     SIDE = 'R'
```

TRANS = 'N': P * C C * P

TRANS = 'C': P**H * C C * P**H

Here Q and P**H are the unitary matrices determined by CGEBRD when reducing a complex matrix A to bidiagonal form: A = Q * B * P**H. Q and P**H are defined as products of elementary reflectors `H(i)` and `G(i)` respectively.

Let nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Thus nq is the order of the unitary matrix Q or P**H that is applied.

If VECT = 'Q', A is assumed to have been an NQ-by-K matrix: if nq >= k, Q = `H(1) H(2)` . . . H(k);

if nq < k, Q = `H(1) H(2)` . . . H(nq-1).

If VECT = 'P', A is assumed to have been a K-by-NQ matrix: if k < nq, P = `G(1) G(2)` . . . G(k);

if k >= nq, P = `G(1) G(2)` . . . G(nq-1).

ARGUMENTS

- **VECT (input)**

  ```
  = 'Q': apply Q or Q**H;

  = 'P': apply P or P**H.
  ```

- **SIDE (input)**

  ```
  = 'L': apply Q, Q**H, P or P**H from the Left;

  = 'R': apply Q, Q**H, P or P**H from the Right.
  ```

- **TRANS (input)**

```
= 'N':  No transpose, apply Q or P;

= 'C':  Conjugate transpose, apply Q**H or P**H.
```

- **M (input)**
  The number of rows of the matrix C. M >= 0.
- **N (input)**
  The number of columns of the matrix C. N >= 0.
- **K (input)**
  If VECT = 'Q', the number of columns in the original matrix reduced by CGEBRD. If VECT = 'P', the number of rows in the original matrix reduced by CGEBRD. K >= 0.
- **A (input)**
  (LDA,min(nq,K)) if VECT = 'Q' (LDA,nq) if VECT = 'P' The vectors which define the elementary reflectors `H(i)` and G(i), whose products determine the matrices Q and P, as returned by CGEBRD.
- **LDA (input)**
  The leading dimension of the array A. If VECT = 'Q', LDA >= max(1,nq); if VECT = 'P', LDA >= max(1,min(nq,K)).
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector `H(i)` or `G(i)` which determines Q or P, as returned by CGEBRD in the array argument TAUQ or TAUP.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q or P*C or P**H*C or C*P or C*P**H.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmhr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMHR( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, ILO, IHI, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMHR_64( SIDE, TRANS, M, N, ILO, IHI, A, LDA, TAU, C,
*      LDC, WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, ILO, IHI, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMHR( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMHR_64( SIDE, [TRANS], [M], [N], ILO, IHI, A, [LDA],
*       TAU, C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, ILO, IHI, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmhr(char side, char trans, int m, int n, int ilo, int ihi, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmhr_64(char side, char trans, long m, long n, long ilo, long ihi, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmhr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of IHI-ILO elementary reflectors, as returned by CGEHRD:

Q = `H(ilo)` `H(ilo+1)` . . . H(ihi-1).

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q**H (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **ILO (input)**
  ILO and IHI must have the same values as in the previous call of CGEHRD. Q is equal to the unit matrix except in the submatrix Q(ilo+1:ihi,ilo+1:ihi). If SIDE = 'L', then 1 < = ILO < = IHI < = M, if M > 0, and ILO = 1 and IHI = 0, if M = 0; if SIDE = 'R', then 1 < = ILO < = IHI < = N, if N > 0, and ILO = 1 and IHI = 0, if N = 0.

- **IHI (input)**
  See the description of ILO.

- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CGEHRD.

- **LDA (input)**
  The leading dimension of the array A. LDA > = `max(1,M)` if SIDE = 'L'; LDA > = `max(1,N)` if SIDE = 'R'.

- **TAU (input)**

(M-1) if SIDE = 'L' (N-1) if SIDE = 'R' TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGEHRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunml2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

---

SYNOPSIS

```
 SUBROUTINE ZUNML2( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, INFO

 SUBROUTINE ZUNML2_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNML2( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, INFO

 SUBROUTINE UNML2_64( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void zunml2(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunml2_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunml2 overwrites the general complex m-by-n matrix C with

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF. Q is of order m if SIDE = 'L' and of order n if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q' from the Left

  = 'R': apply Q or Q' from the Right
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q' (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**

TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.

- **C (input/output)**
  On entry, the m-by-n matrix C. On exit, C is overwritten by Q*C or Q'*C or C*Q' or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L', (M) if SIDE = 'R'
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmlq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMLQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMLQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMLQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMLQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmlq(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmlq_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmlq overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k)' . . . H(2)' H(1)'
```

as returned by CGELQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGELQF in the first k rows of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**

TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGELQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmql - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMQL( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMQL_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMQL( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMQL_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmql(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmql_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmql overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(k) . . . H(2) H(1)
```

as returned by CGEQLF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Transpose, apply Q**H.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQLF in the last k columns of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).

- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQLF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmqr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMQR( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMQR_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMQR( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMQR_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmqr(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmqr_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmqr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by CGEQRF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```

- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  The i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGEQRF in the first k columns of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. If SIDE = 'L', LDA > = max(1,M); if SIDE = 'R', LDA > = max(1,N).

- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CGEQRF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmr2 - overwrite the general complex m-by-n matrix C with Q * C if SIDE = 'L' and TRANS = 'N', or Q'* C if SIDE = 'L' and TRANS = 'C', or C * Q if SIDE = 'R' and TRANS = 'N', or C * Q' if SIDE = 'R' and TRANS = 'C',

---

SYNOPSIS

```
 SUBROUTINE ZUNMR2( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, INFO

 SUBROUTINE ZUNMR2_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMR2( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, INFO

 SUBROUTINE UNMR2_64( SIDE, TRANS, [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmr2(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmr2_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmr2 overwrites the general complex m-by-n matrix C with

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF. Q is of order m if SIDE = 'L' and of order n if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q' from the Left

  = 'R': apply Q or Q' from the Right
  ```

- **TRANS (input)**

  ```
  = 'N': apply Q  (No transpose)

  = 'C': apply Q' (Conjugate transpose)
  ```

- **M (input)**
  The number of rows of the matrix C. M > = 0.

- **N (input)**
  The number of columns of the matrix C. N > = 0.

- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.

- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.

- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).

- **TAU (input)**

TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.

- **C (input/output)**
  On entry, the m-by-n matrix C. On exit, C is overwritten by Q*C or Q'*C or C*Q' or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L', (M) if SIDE = 'R'
- **INFO (output)**

```
= 0: successful exit

< 0: if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmrq - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMRQ( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC, WORK,
*      LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMRQ_64( SIDE, TRANS, M, N, K, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMRQ( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMRQ_64( SIDE, [TRANS], [M], [N], [K], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmrq(char side, char trans, int m, int n, int k, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmrq_64(char side, char trans, long m, long n, long k, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmrq overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1)' H(2)' . . . H(k)'
```

as returned by CGERQF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CGERQF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**
  The leading dimension of the array A. LDA > = max(1,K).
- **TAU (input)**

TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CGERQF.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zunmrz - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, K, L, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, K, L, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE ZUNMRZ( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*       WORK, LWORK, INFO)
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, K, L, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMRZ_64( SIDE, TRANS, M, N, K, L, A, LDA, TAU, C, LDC,
*       WORK, LWORK, INFO)
 CHARACTER(LEN=1) :: SIDE, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, K, L, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmrz(char side, char trans, int m, int n, int k, int l, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, doublecomplex *work, int lwork, int *info);

void zunmrz_64(char side, char trans, long m, long n, long k, long l, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, doublecomplex *work, long lwork, long *info);

PURPOSE

zunmrz overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix defined as the product of k elementary reflectors

```
Q = H(1) H(2) . . . H(k)
```

as returned by CTZRZF. Q is of order M if SIDE = 'L' and of order N if SIDE = 'R'.

ARGUMENTS

- **SIDE (input)**

  ```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
  ```
- **TRANS (input)**

  ```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
  ```
- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **K (input)**
  The number of elementary reflectors whose product defines the matrix Q. If SIDE = 'L', M > = K > = 0; if SIDE = 'R', N > = K > = 0.
- **L (input)**
  The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If SIDE = 'L', M > = L > = 0, if SIDE = 'R', N > = L > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L', (LDA,N) if SIDE = 'R' The i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by CTZRZF in the last k rows of its array argument A. A is modified by the routine but restored on exit.
- **LDA (input)**

The leading dimension of the array A. LDA >= max(1,K).

- **TAU (input)**
  TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CTZRZF.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (output)**
  On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >= M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

FURTHER DETAILS

Based on contributions by

```
A. Petitet, Computer Science Dept., Univ. of Tenn., Knoxville, USA
```

NAME

zunmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUNMTR( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, LDA, LDC, LWORK, INFO

 SUBROUTINE ZUNMTR_64( SIDE, UPLO, TRANS, M, N, A, LDA, TAU, C, LDC,
*      WORK, LWORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 DOUBLE COMPLEX A(LDA,*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, LDA, LDC, LWORK, INFO
```

F95 INTERFACE

```
 SUBROUTINE UNMTR( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU, C,
*       [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER :: M, N, LDA, LDC, LWORK, INFO

 SUBROUTINE UNMTR_64( SIDE, UPLO, [TRANS], [M], [N], A, [LDA], TAU,
*       C, [LDC], [WORK], [LWORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX(8), DIMENSION(:) :: TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: A, C
 INTEGER(8) :: M, N, LDA, LDC, LWORK, INFO
```

C INTERFACE

#include <sunperf.h>

void zunmtr(char side, char uplo, char trans, int m, int n, doublecomplex *a, int lda, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zunmtr_64(char side, char uplo, char trans, long m, long n, doublecomplex *a, long lda, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zunmtr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by CHETRD:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
 = 'L': apply Q or Q**H from the Left;

 = 'R': apply Q or Q**H from the Right.
```

- **UPLO (input)**

```
 = 'U': Upper triangle of A contains elementary reflectors
from CHETRD;
 = 'L': Lower triangle of A contains elementary reflectors
from CHETRD.
```

- **TRANS (input)**

```
 = 'N':  No transpose, apply Q;

 = 'C':  Conjugate transpose, apply Q**H.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **A (input)**
  (LDA,M) if SIDE = 'L' (LDA,N) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CHETRD.
- **LDA (input)**

The leading dimension of the array A. LDA >= `max(1,M)` if SIDE = 'L'; LDA >= `max(1,N)` if SIDE = 'R'.

- **TAU (input)**
  (M-1) if SIDE = 'L' (N-1) if SIDE = 'R' `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHETRD.
- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  On exit, if INFO = 0, `WORK(1)` returns the optimal LWORK.
- **LWORK (input)**
  The dimension of the array WORK. If SIDE = 'L', LWORK >= max(1,N); if SIDE = 'R', LWORK >= max(1,M). For optimum performance LWORK >= N*NB if SIDE = 'L', and LWORK >=M*NB if SIDE = 'R', where NB is the optimal blocksize.

  If LWORK = -1, then a workspace query is assumed; the routine only calculates the optimal size of the WORK array, returns this value as the first entry of the WORK array, and no error message related to LWORK is issued by XERBLA.
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zupgtr - generate a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by CHPTRD using packed storage

---

SYNOPSIS

```
SUBROUTINE ZUPGTR( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), TAU(*), Q(LDQ,*), WORK(*)
INTEGER N, LDQ, INFO

SUBROUTINE ZUPGTR_64( UPLO, N, AP, TAU, Q, LDQ, WORK, INFO)
CHARACTER * 1 UPLO
DOUBLE COMPLEX AP(*), TAU(*), Q(LDQ,*), WORK(*)
INTEGER*8 N, LDQ, INFO
```

F95 INTERFACE

```
SUBROUTINE UPGTR( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: Q
INTEGER :: N, LDQ, INFO

SUBROUTINE UPGTR_64( UPLO, [N], AP, TAU, Q, [LDQ], [WORK], [INFO])
CHARACTER(LEN=1) :: UPLO
COMPLEX(8), DIMENSION(:) :: AP, TAU, WORK
COMPLEX(8), DIMENSION(:,:) :: Q
INTEGER(8) :: N, LDQ, INFO
```

C INTERFACE

#include <sunperf.h>

void zupgtr(char uplo, int n, doublecomplex *ap, doublecomplex *tau, doublecomplex *q, int ldq, int *info);

void zupgtr_64(char uplo, long n, doublecomplex *ap, doublecomplex *tau, doublecomplex *q, long ldq, long *info);

PURPOSE

zupgtr generates a complex unitary matrix Q which is defined as the product of n-1 elementary reflectors `H(i)` of order n, as returned by CHPTRD using packed storage:

if UPLO = 'U', Q = `H(n-1)` . . . `H(2)` H(1),

if UPLO = 'L', Q = `H(1)` `H(2)` . . . H(n-1).

ARGUMENTS

- **UPLO (input)**

```
 = 'U': Upper triangular packed storage used in previous
call to CHPTRD;
 = 'L': Lower triangular packed storage used in previous
call to CHPTRD.
```

- **N (input)**
  The order of the matrix Q. N > = 0.
- **AP (input)**
  The vectors which define the elementary reflectors, as returned by CHPTRD.
- **TAU (input)**
  `TAU(i)` must contain the scalar factor of the elementary reflector H(i), as returned by CHPTRD.
- **Q (output)**
  The N-by-N unitary matrix Q.
- **LDQ (input)**
  The leading dimension of the array Q. LDQ > = max(1,N).
- **WORK (workspace)**
  `dimension(N-1)`
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zupmtr - overwrite the general complex M-by-N matrix C with SIDE = 'L' SIDE = 'R' TRANS = 'N'

SYNOPSIS

```
 SUBROUTINE ZUPMTR( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC, WORK,
*      INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 DOUBLE COMPLEX AP(*), TAU(*), C(LDC,*), WORK(*)
 INTEGER M, N, LDC, INFO

 SUBROUTINE ZUPMTR_64( SIDE, UPLO, TRANS, M, N, AP, TAU, C, LDC,
*      WORK, INFO)
 CHARACTER * 1 SIDE, UPLO, TRANS
 DOUBLE COMPLEX AP(*), TAU(*), C(LDC,*), WORK(*)
 INTEGER*8 M, N, LDC, INFO
```

F95 INTERFACE

```
 SUBROUTINE UPMTR( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX(8), DIMENSION(:) :: AP, TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: C
 INTEGER :: M, N, LDC, INFO

 SUBROUTINE UPMTR_64( SIDE, UPLO, [TRANS], [M], [N], AP, TAU, C, [LDC],
*       [WORK], [INFO])
 CHARACTER(LEN=1) :: SIDE, UPLO, TRANS
 COMPLEX(8), DIMENSION(:) :: AP, TAU, WORK
 COMPLEX(8), DIMENSION(:,:) :: C
 INTEGER(8) :: M, N, LDC, INFO
```

C INTERFACE

#include <sunperf.h>

void zupmtr(char side, char uplo, char trans, int m, int n, doublecomplex *ap, doublecomplex *tau, doublecomplex *c, int ldc, int *info);

void zupmtr_64(char side, char uplo, char trans, long m, long n, doublecomplex *ap, doublecomplex *tau, doublecomplex *c, long ldc, long *info);

PURPOSE

zupmtr overwrites the general complex M-by-N matrix C with TRANS = 'C': Q**H * C C * Q**H

where Q is a complex unitary matrix of order nq, with nq = m if SIDE = 'L' and nq = n if SIDE = 'R'. Q is defined as the product of nq-1 elementary reflectors, as returned by CHPTRD using packed storage:

if UPLO = 'U', Q = `H(nq-1) . . . H(2)` H(1);

if UPLO = 'L', Q = `H(1) H(2)` . . . H(nq-1).

ARGUMENTS

- **SIDE (input)**

```
  = 'L': apply Q or Q**H from the Left;

  = 'R': apply Q or Q**H from the Right.
```

- **UPLO (input)**

```
  = 'U': Upper triangular packed storage used in previous
call to CHPTRD;
  = 'L': Lower triangular packed storage used in previous
call to CHPTRD.
```

- **TRANS (input)**

```
  = 'N':  No transpose, apply Q;

  = 'C':  Conjugate transpose, apply Q**H.
```

- **M (input)**
  The number of rows of the matrix C. M > = 0.
- **N (input)**
  The number of columns of the matrix C. N > = 0.
- **AP (input)**
  (M*(M+1)/2) if SIDE = 'L' (N*(N+1)/2) if SIDE = 'R' The vectors which define the elementary reflectors, as returned by CHPTRD. AP is modified by the routine but restored on exit.
- **TAU (input)**

or (N-1) if SIDE = 'R' TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by CHPTRD.

- **C (input/output)**
  On entry, the M-by-N matrix C. On exit, C is overwritten by Q*C or Q**H*C or C*Q**H or C*Q.
- **LDC (input)**
  The leading dimension of the array C. LDC >= max(1,M).
- **WORK (workspace)**
  (N) if SIDE = 'L' (M) if SIDE = 'R'
- **INFO (output)**

```
= 0:  successful exit

< 0:  if INFO  = -i, the i-th argument had an illegal value
```

NAME

zvmul - compute the scaled product of complex vectors

SYNOPSIS

```
SUBROUTINE ZVMUL( N, ALPHA, X, INCX, Y, INCY, BETA, Z, INCZ)
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX X(*), Y(*), Z(*)
INTEGER N, INCX, INCY, INCZ

SUBROUTINE ZVMUL_64( N, ALPHA, X, INCX, Y, INCY, BETA, Z, INCZ)
DOUBLE COMPLEX ALPHA, BETA
DOUBLE COMPLEX X(*), Y(*), Z(*)
INTEGER*8 N, INCX, INCY, INCZ
```

F95 INTERFACE

```
SUBROUTINE VMUL( [N], ALPHA, X, [INCX], Y, [INCY], BETA, Z, [INCZ])
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: X, Y, Z
INTEGER :: N, INCX, INCY, INCZ

SUBROUTINE VMUL_64( [N], ALPHA, X, [INCX], Y, [INCY], BETA, Z, [INCZ])
COMPLEX(8) :: ALPHA, BETA
COMPLEX(8), DIMENSION(:) :: X, Y, Z
INTEGER(8) :: N, INCX, INCY, INCZ
```

C INTERFACE

#include <sunperf.h>

void zvmul(int n, doublecomplex alpha, doublecomplex *x, int incx, doublecomplex *y, int incy, doublecomplex beta, doublecomplex *z, int incz);

void zvmul_64(long n, doublecomplex alpha, doublecomplex *x, long incx, doublecomplex *y, long incy, doublecomplex beta, doublecomplex *z, long incz);

PURPOSE

zvmul computes the scaled product of complex vectors:

```
z(i) = ALPHA * x(i) * y(i) + BETA * z(i)
```

for 1 <= i <= N.

ARGUMENTS

- **N (input)**
  Length of the vectors. N > = 0. ZVMUL will return immediately if N = 0.
- **ALPHA (input)**
  Scale factor on the multiplicand vectors.
- **X (input)**

  ```
  dimension(*)
  ```

  Multiplicand vector.
- **INCX (input)**
  Stride between elements of the multiplicand vector X. INCX > 0.
- **Y (input)**

  ```
  dimension(*)
  ```

  Multiplicand vector.
- **INCY (input)**
  Stride between elements of the multiplicand vector Y. INCY > 0.
- **BETA (input)**
  Scale factor on the product vector.
- **Z (input/output)**

  ```
  dimension(*)
  ```

  Product vector. On exit, `z(i)` = ALPHA * `x(i)` * `y(i)` + BETA * z(i).
- **INCZ (input)**
  Stride between elements of Z. INCZ > 0.