Sun Performance Library User's Guide for Fortran and C

Forte Developer 6 update 2
(Sun WorkShop 6 update 2)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-7995-10
July 2001, Revision A

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

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	**New Product Name**
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Tables

Before You Begin

This book describes how to use the Sun™ specific extensions and features included with the Sun Performance Library™ subroutines that are supported by the Sun WorkShop™ 6 FORTRAN 77, Fortran 95, and C compilers.

Who Should Use This Book

This book is a user's guide intended for programmers who have a working knowledge of the Fortran or C language and some understanding of the base LAPACK, BLAS, FFTPACK, VFFTPACK, and LINPACK libraries available from Netlib (`http://www.netlib.org`).

How This Book Is Organized

This book is organized into the following chapters and appendixes:

Chapter 1, "Introduction," describes the benefits of using the Sun Performance Library and the features of the Sun Performance Library.

Chapter 2, "Using Sun Performance Library," describes how to use the `f77`, `f95`, and C interfaces provided with the Sun Performance Library.

Chapter 3, "SPARC Optimization and Parallel Processing," shows how to use compiler and linking options to maximize library performance for specific SPARC™ instruction set architectures and different parallel processing modes.

Chapter 4, “Working With Matrices,” includes information on matrix storage schemes, matrix types, and sparse matrices.

Chapter 5, “Using Sun Performance Library Fast Fourier Transform Routines,” describes the one-dimensional, two-dimensional, and three-dimensional fast Fourier transform routines provided with the Sun Performance Library.

Chapter 6, “Using Sun Performance Library Convolution and Correlation Routines,” provides examples of using the convolution and correlation routines provided with the Sun Performance Library.

Appendix A, “Sun Performance Library Routines,” lists the Sun Performance Library routines organized according to name, routine, and library.

What Is Not in This Book

This book does not repeat information included in existing LAPACK books or sources on Netlib. Refer to the section “Related Documents and Web Sites” for a list of sources that contain reference material for the base routines upon which Sun Performance Library is based.

Related Documents and Web Sites

A number of books and web sites provide reference information on the routines in the base libraries (LAPACK, LINPACK, BLAS, and so on) upon which the Sun Performance Library is based. The following books augment this manual and provide essential information:

- *LAPACK Users’ Guide*. 3rd ed., Anderson E. and others. SIAM, 1999.
- *LINPACK User’s Guide*. Dongarra J. J. and others. SIAM, 1979.

The *LAPACK Users’ Guide*, 3rd ed. is the official reference for the base LAPACK version 3.0 routines. An online version of the *LAPACK 3.0 Users’ Guide* is available at `http://www.netlib.org/lapack/lug/`, and the printed version is available from the Society for Industrial and Applied Mathematics (SIAM) `http://www.siam.org`.

Sun Performance Library routines contain performance enhancements, extensions, and features not described in the *LAPACK Users’ Guide*. However, because Sun Performance Library maintains compatibility with the base LAPACK routines, the *LAPACK Users’ Guide* can be used as a reference for the LAPACK routines and the FORTRAN 77 interfaces.

Online Resources

Online information describing the performance library routines that form the basis of the Sun Performance Library can be found at the following URLs.

LAPACK version 3.0	`http://www.netlib.org/lapack/`
BLAS, levels 1 through 3	`http://www.netlib.org/blas/`
FFTPACK version 4	`http://www.netlib.org/fftpack/`
VFFTPACK version 2.1	`http://www.netlib.org/vfftpack/`
Sparse BLAS	`http://www.netlib.org/sparse-blas/index.html`
NIST (National Institute of Standards and Technology) Fortran Sparse BLAS	`http://math.nist.gov/spblas/`
LINPACK	`http://www.netlib.org/linpack/`

Typographic Conventions

Typeface	**Meaning**	**Examples**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	`%`
Bourne shell and Korn shell	`$`
C shell, Bourne shell, and Korn shell superuser	`#`

Supported Platforms

This Sun WorkShop™ Sun Performance Library release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* operating environment.

Accessing Sun WorkShop Development Tools and Man Pages

The Sun WorkShop product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Sun WorkShop compilers and tools, you must have the Sun WorkShop component directory in your `PATH` environment variable. To access the Sun WorkShop man pages, you must have the Sun WorkShop man page directory in your `MANPATH` environment variable.

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` and `MANPATH` variables to access this release, see the *Sun WorkShop 6 update 2 Installation Guide* or your system administrator.

Note – The information in this section assumes that your Sun WorkShop 6 update 2 products are installed in the `/opt` directory. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing Sun WorkShop Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the Sun WorkShop compilers and tools.

To Determine If You Need to Set Your `PATH` Environment Variable

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing `/opt/SUNWspro/bin/`.**

   If you find the path, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the path, set your `PATH` environment variable by following the instructions in the next section.

To Set Your `PATH` Environment Variable to Enable Access to Sun WorkShop Compilers and Tools

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**

2. **Add the following to your `PATH` environment variable.**

   `/opt/SUNWspro/bin`

Accessing Sun WorkShop Man Pages

Use the following steps to determine whether you need to change your `MANPATH` variable to access the Sun WorkShop `man` pages.

To Determine If You Need to Set Your `MANPATH` Environment Variable

1. **Request the `workshop` man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

   If the `workshop`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next section for setting your `MANPATH` environment variable.

To Set Your `MANPATH` Environment Variable to Enable Access to Sun WorkShop Man Pages

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `MANPATH` environment variable.**

   `/opt/SUNWspro/man`

Accessing Sun WorkShop Documentation

You can access Sun WorkShop product documentation at the following locations:

- **The product documentation is available from the documentation index installed with the product on your local system or network.**

  Point your Netscape™ Communicator 4.0 or compatible Netscape version browser to the following file:

  `/opt/SUNWspro/docs/index.html`

  If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- **Manuals are available from the docs.sun.com^{sm} Web site.**

  The `docs.sun.com` Web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris 8 Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris 8 Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris 8 Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Ordering Sun Documentation

You can order product documentation directly from Sun through the `docs.sun.com` Web site or from Fatbrain.com, an Internet bookstore. You can find the Sun Documentation Center on Fatbrain.com at the following URL:

`http://www.fatbrain.com/documentation/sun`

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

CHAPTER 1

Introduction

Sun Performance Library is a set of optimized, high-speed mathematical subroutines for solving linear algebra and other numerically intensive problems. Sun Performance Library is based on a collection of public domain applications available from Netlib at `http://www.netlib.org`. Sun has enhanced these public domain applications and bundled them as the Sun Performance Library.

The *Sun Performance Library User's Guide* explains the Sun-specific enhancements to the base applications available from Netlib. Reference material describing the base routines is available from Netlib and the Society for Industrial and Applied Mathematics (SIAM).

Libraries Included With Sun Performance Library

Sun Performance Library contains enhanced versions of the following standard libraries:

- LAPACK version 3.0 – For solving linear algebra problems.
- BLAS1 (Basic Linear Algebra Subprograms) – For performing vector-vector operations.
- BLAS2 – For performing matrix-vector operations.
- BLAS3 – For performing matrix-matrix operations.
- FFTPACK version 4 – For performing the fast Fourier transform.
- VFFTPACK version 2.1 – A vectorized version of FFTPACK for performing the fast Fourier transform.
- LINPACK – For solving linear algebra problems in legacy applications containing routines that have not been upgraded to LAPACK 3.0.

Note – LAPACK version 3.0 supersedes LINPACK and all previous versions of LAPACK. Use LAPACK for new development and LINPACK to support legacy applications.

Sun Performance Library is available in both static and dynamic library versions optimized for the V8, V8+, and V9 architectures. Sun Performance Library supports static and shared libraries on Solaris 2.6, Solaris 7, and Solaris 8 and adds support for multiple processors.

Sun Performance Library LAPACK routines have been compiled with a Fortran 95 compiler and remain compatible with the Netlib LAPACK version 3.0 library. The Sun Performance Library versions of these routines perform the same operations as the Fortran callable routines and have the same interface as the standard Netlib versions.

LAPACK contains driver, computational, and auxiliary routines. Sun Performance Library does not support the auxiliary routines, because auxiliary routines can change or be removed from LAPACK without notice. Because the auxiliary routines are not supported, they are not documented in the Sun Performance Library User's Guide or the section 3P man pages.

Many auxiliary routines contain LA as the second and third characters in the routine name; however, some do not. Appendix B of the *LAPACK Users' Guide* contains a list of auxiliary routines.

Auxiliary routines are not available in the shared (dynamic) libraries, but the auxiliary routines are still available in the static libraries. However, there is no guarantee that auxiliary routines will continue to be available in any form in future versions of the Sun Performance Library.

Netlib

Netlib is an online repository of mathematical software, papers, and databases maintained by AT&T Bell Laboratories, the University of Tennessee, Oak Ridge National Laboratory, and professionals from around the world.

Netlib provides many libraries, in addition to the libraries used in Sun Performance Library. While some of these libraries can appear similar to libraries used with Sun Performance Library, they can be different from, and incompatible with Sun Performance Library.

Using routines from other libraries can produce compatibility problems, not only with Sun Performance Library routines, but also with the base Netlib LAPACK routines. When using routines from other libraries, refer to the documentation provided with those libraries.

For example, Netlib provides a CLAPACK library, but the CLAPACK interfaces differ from the C interfaces included with Sun Performance Library. A LAPACK 90 library package is also available on Netlib. The LAPACK 90 library contains interfaces that differ from the Sun Performance Library Fortran 95 interfaces and the Netlib LAPACK version 3.0 interfaces. If using LAPACK 90, refer to the documentation provided with that library.

For the base libraries supported by Sun Performance Library, Netlib provides detailed information that can supplement this user's guide. The *LAPACK 3.0 Users' Guide* describes LAPACK algorithms and how to use the routines, but it does not describe the Sun Performance Library extensions made to the base routines.

Sun Performance Library Features

Sun Performance Library routines can increase application performance on both serial and MP platforms, because the serial speed of many Sun Performance Library routines has been increased, and many routines have been parallelized that might be serial in other products. Sun Performance Library routines also have SPARC specific optimizations that are not present in the base Netlib libraries.

Sun Performance Library provides the following optimizations and extensions to the base Netlib libraries:

- Extensions that support Fortran 95 and C language interfaces
- Fortran 95 language features, including type independence, compile time checking, and optional arguments.
- Consistent API across the different libraries in Sun Performance Library
- Compatibility with LAPACK 1.x, LAPACK 2.0, and LAPACK 3.0 libraries
- Increased performance, and in some cases, greater accuracy
- Optimizations for specific SPARC instruction set architectures
- Support for 64-bit enabled Solaris operating environment
- Support for parallel processing compiler options
- Support for multiple processor hardware options

Mathematical Routines

The Sun Performance Library routines are used to solve the following types of linear algebra and numerical problems:

- *Elementary vector and matrix operations* – Vector and matrix products; plane rotations; 1, 2-, and infinity-norms; rank-1, 2, k, and 2k updates
- *Linear systems* – Solve full-rank systems, compute error bounds, solve Sylvester equations, refine a computed solution, equilibrate a coefficient matrix
- *Least squares* – Full-rank, generalized linear regression, rank-deficient, linear equality constrained
- *Eigenproblems* – Eigenvalues, generalized eigenvalues, eigenvectors, generalized eigenvectors, Schur vectors, generalized Schur vectors
- *Matrix factorizations or decompositions* – SVD, generalized SVD, QL and LQ, QR and RQ, Cholesky, LU, Schur, LDL^T and UDU^T
- *Support operations* – Condition number, in-place or out-of-place transpose, inverse, determinant, inertia
- *Sparse matrices* – Solve symmetric, structurally symmetric, and unsymmetric coefficient matrices using direct methods and a choice of fill-reducing ordering algorithms, and user-specified orderings
- Convolution and correlation in one and two dimensions
- Fast Fourier transforms, Fourier synthesis, cosine and quarter-wave cosine transforms, cosine and quarter-wave sine transforms
- Complex vector FFTs and FFTs in two and three dimensions

Compatibility With Previous LAPACK Versions

The Sun Performance Library routines that are based on LAPACK support the expanded capabilities and improved algorithms in LAPACK 3.0, but are completely compatible with both LAPACK l.x and LAPACK 2.0. Maintaining compatibility with previous LAPACK versions:

- Reduces linking errors due to changes in subroutine names or argument lists.
- Ensures results are consistent with results generated with previous LAPACK versions.
- Minimizes programs terminating due to differences between argument lists.

Getting Started With Sun Performance Library

This section shows the most basic compiler options used to compile an application that uses the Sun Performance Library routines.

To use the Sun Performance Library, type one of the following commands.

```
my_system% f95 -dalign my_file.f -xlic_lib=sunperf
```

or

```
my_system% cc -dalign my_file.c -xlic_lib=sunperf
```

Because Sun Performance Library routines are compiled with `-dalign`, the `-dalign` option should be used for compilation of all files if any routine in the program makes a Sun Performance Library call. If `-dalign` cannot be used, enabling Trap 6, described in the section "Enabling Trap 6" on page 14, is a low-performance workaround that allows misaligned data.

Sun Performance Library is linked into an application with the `-xlic_lib` switch rather than the `-l` switch that is used to link in other libraries. The `-xlic_lib` switch gives the same effect as if `-l` was used to specify the Sun Performance Library and added `-l` switches for all of the supporting libraries that Sun Performance Library requires.

To summarize, use the following:

- `-dalign` on all files at compile time or enable trap 6
- The same command line options for compiling and linking
- `-xlic_lib=sunperf`

Additional compiler options exist that optimize application performance for the following:

- Specific SPARC instruction set architectures, as described in "Compiling for SPARC Platforms" on page 28.
- Parallel processing, as described in "Parallel Processing" on page 33.

Enabling Trap 6

If an application cannot be compiled using -dalign, enable trap 6 to provide a handler for misaligned data. To enable trap 6 on SPARC, do the following:

1. **Place this assembly code in a file called trap6_handler.s.**

```
    .global trap6_handler_
    .text
    .align 4
trap6_handler_:
    retl
    ta    6
```

2. **Assemble trap6_handler.s.**

   my_system% **fbe trap6_handler.s**

   The first parallelizable subroutine invoked from Sun Performance Library will call a routine named trap6_handler_. If a trap6_handler_ is not specified, Sun Performance Library will call a default handler that does nothing. Not supplying a handler for any misaligned data will cause a trap that will be fatal. (fbe (1) is the Solaris assembler for SPARC platforms.)

3. **Include trap6_handler.o on the command line.**

```
my_system% f95 any.f trap6_handler.o -xlic_lib=sunperf
```

CHAPTER 2

Using Sun Performance Library

This chapter describes using the Sun Performance Library to improve the execution speed of applications written in FORTRAN 77, Fortran 95, or C. The performance of many applications can be increased by using Sun Performance Library without making source code changes or recompiling. However, some modifications to applications might be required to gain peak performance with Sun Performance Library.

Improving Application Performance

The following sections describe ways of using Sun Performance Library routines without making source code changes or recompiling.

Replacing Routines With Sun Performance Library Routines

Many applications use one or more of the base Netlib libraries, such as LAPACK or BLAS. Because Sun Performance Library maintains the same interfaces and functionality of these libraries, base Netlib routines can be replaced with Sun Performance Library routines. Application performance is increased, because Sun Performance Library routines can be faster than the corresponding Netlib routines or similar routines provided by other vendors.

Improving Performance of Other Libraries

Many commercial math libraries are built around a core of generic BLAS and LAPACK routines. When an application has a dependency on proprietary interfaces in another library that prevents the library from being completely replaced, the BLAS and LAPACK routines used in that library can be replaced with the Sun Performance Library BLAS and LAPACK routines. Because replacing the core routines does not require any code changes, the proprietary library features can still be used, and the other routines in the library can remain unchanged.

Using Tools to Restructure Code

Some libraries that do not directly use Sun Performance Library routines can be modified by using automatic code restructuring tools that replace existing code with Sun Performance Library code. For example, a source- to- source conversion tool can replace existing BLAS code structures with calls to the Sun Performance Library BLAS routines. These conversion tools can also recognize many user written matrix multiplications and replace them with calls to the matrix multiplication subroutine in Sun Performance Library.

Fortran `f77/f95` Interfaces

Sun Performance Library `f77/f95` interfaces use the following conventions:

- All arguments are passed by reference.
- Types of arguments must be consistent within a call (For example, do not mix `REAL*8` and `REAL*4` parameters in the same call.
- Arrays are stored columnwise.
- Indices are based at one, in keeping with standard Fortran practice.

When calling Sun Performance Library routines:

- Do not prototype the subroutines with the Fortran 95 `INTERFACE` statement. Use the `USE SUNPERF` statement instead.
- Do not use `-ext_names=plain` to compile routines that call routines from Sun Performance Library.

Fortran SUNPERF Module for Use with Fortran 95

Sun Performance Library provides a Fortran module for additional ease-of-use features with Fortran 95 programs. To use this module, include the following line in Fortran 95 codes.

```
USE SUNPERF
```

USE statements must precede all other statements in the code, except for the PROGRAM or SUBROUTINE statement.

The SUNPERF module contains interfaces that simplify the calling sequences and provides the following features:

- *Type Independence* – Sun Performance Library supports interfaces where the type of the data arguments will automatically be recognized, eliminating the need for type-dependent prefixes (S, D, C, or Z). In the FORTRAN 77 routines, the type must be specified as part of the routine name. For example, DGEMM is a double precision matrix multiply and SGEMM is a single precision matrix multiply. When calling GEMM with the Fortran 95 interfaces, Fortran will infer the type from the arguments that are passed. Passing single-precision arguments to GEMM gets results that are equivalent to specifying SGEMM, and passing double-precision arguments gets results that are equivalent to DGEMM. For example, CALL DSCAL(20,5.26D0,X,1) could be changed to CALL SCAL(20, 5.26D0, X, 1).
- *Compile-Time Checking* – In FORTRAN 77, it is generally impossible for the compiler to determine what arguments should be passed to a particular routine. In Fortran 95, the USE SUNPERF statement allows the compiler to determine the number, type, size, and shape of each argument to each Sun Performance Library routine. It can check the calls against the expected value and display errors during compilation.
- *Optional Arguments* – Sun Performance Library supports interfaces where some arguments are optional. In FORTRAN 77, all arguments must be specified in the order determined by the interface for all routines. All interfaces will support f95 style OPTIONAL attributes on arguments that are not required. Using routines with optional arguments, such as GEMM, are useful for new development. Specifically named routines, such as DGEMM, are maintained to support legacy code. To determine the optional arguments for a routine, refer to the section 3P man pages. In the section 3P man pages, optional arguments are enclosed in square brackets [ ].

- *64-bit Integer Support–* When using the 64-bit interfaces provided with Sun Performance Library, integer arguments need to be promoted to 64-bits, and the routine name needs to be modified by appending _64 to the routine name. With the SUNPERF module, 64-bit integers will automatically be recognized, which eliminates the need for appending _64 to the routine name, as shown in the following code example.

```
SUBROUTINE SUB(N,ALPHA,X,Y)
USE SUNPERF
INTEGER(8) N
REAL(8) ALPHA, X(N), Y(N)

! EQUIVALENT TO DAXPY_64(N,ALPHA,X,1_8,Y,1_8)
CALL DAXPY(N,ALPHA,X,1_8,Y,1_8)

END
```

  When using Sun Performance Library routines with optional arguments, the _64 suffix is required for 64-bit integers, as shown in the following code example.

```
SUBROUTINE SUB(N,ALPHA,X,Y)
USE SUNPERF
INTEGER(8) N
REAL(8) ALPHA, X(N), Y(N)

! EQUIVALENT TO DAXPY_64(N,ALPHA,X,1_8,Y,1_8)
CALL AXPY_64(ALPHA=ALPHA,X=X,Y=Y)

END
```

For a detailed description of using the Sun Performance Library 64-bit interfaces, see "Compiling Code for a 64-Bit Enabled Solaris Operating Environment" on page 29.

Because the `sunperf.mod` file is compiled with `-dalign`, any code that contains the `USE SUNPERF` statement must be compiled with `-dalign`. The following error occurs if the code is not compiled with `-dalign`.

```
 use sunperf
             ^
   "test_code.f", Line = 2, Column = 11: ERROR: Procedure "SUNPERF"
and this compilation must both be compiled with -a dalign, or
without -a dalign.
```

Optional Arguments

Sun Performance Library routines support Fortran 95 optional arguments, where argument values that can be inferred from other arguments can be omitted. For example, the SAXPY routine is defined as follows in the man page.

```
SUBROUTINE SAXPY([N], ALPHA, X, [INCX], Y, [INCY])
REAL ALPHA
INTEGER INCX, INCY, N
REAL X(*), Y(*)
```

The N, INCX, and INCY arguments are optional. Note the square bracket notation in the man pages that denotes the optional arguments.

Suppose the user tries to call the SAXPY routine with the following arguments.

```
USE SUNPERF
COMPLEX ALPHA
REAL    X(100), Y(100), XA(100,100), RALPHA
INTEGER INCX, INCY
```

If mismatches in the type, shape, or number of arguments occur, the compiler would issue the following error message:

```
ERROR: No specific match can be found for the generic subprogram
call "AXPY".
```

Using the arguments defined above, the following examples show incorrect calls to the SAXPY routine due type, shape, or number mismatches.

- *Incorrect type of the arguments*–If SAXPY is called as follows:

  ```
      CALL AXPY(100, ALPHA, X, INCX, Y, INCY)
  ```

  A compiler error occurs because mixing parameter types, such as COMPLEX ALPHA and REAL X, is not supported.

- *Incorrect shape of the arguments*– If SAXPY is called as follows:

  ```
      CALL AXPY(N, RALPHA, XA, INCX, Y, INCY)
  ```

  A compiler error occurs because the XA argument is two dimensional, but the interface is expecting a one-dimensional argument.

- *Incorrect number of arguments*– If SAXPY is called as follows:

  ```
      CALL AXPY(RALPHA, X, INCX, Y)
  ```

  A compiler error occurs because the compiler cannot find a routine in the AXPY interface group that takes four arguments of the following form.

  ```
      AXPY(REAL, REAL 1-D ARRAY, INTEGER, REAL 1-D ARRAY)
  ```

  In the following example, the f95 keyword parameter passing capability can allow a user to make essentially the same call using that capability.

  ```
      CALL AXPY(ALPHA=RALPHA,X=X,INCX=INCX,Y=Y)
  ```

  This is a valid call to the AXPY interface. It is necessary to use keyword parameter passing on any parameter that appears in the list *after* the first OPTIONAL parameter is omitted.

The following calls to the AXPY interface are valid.

```
    CALL AXPY(N,RALPHA,X,Y=Y,INCY=INCY)
    CALL AXPY(N,RALPHA,X,INCX,Y)
    CALL AXPY(N,RALPHA,X,Y=Y)
    CALL AXPY(ALPHA=RALPHA,X=X,Y=Y)
```

Fortran Examples

To increase the performance of single processor applications, identify code constructs in an application that can be replaced by calls to Sun Performance Library routines. Performance of multiprocessor applications can increased by identifying opportunities for parallelization.

To increase application performance by modifying code to use Sun Performance Library routines, identify blocks of code that exactly duplicate the capability of a Sun Performance Library routine. The following code example is the matrix-vector product $y \leftarrow Ax + y$, which can be replaced with the `DGEMV` subroutine.,

```
       DO I = 1, N
           DO J = 1, N
               Y(I) = Y(I) + A(I,J) * X(J)
           END DO
       END DO
```

In other cases, a block of code can be equivalent to several Sun Performance Library calls or contain portions of code that can be replaced with calls to Sun Performance Library routines. Consider the following code example.

```
       DO I = 1, N
           IF (V2(I,K) .LT. 0.0) THEN
               V2(I,K) = 0.0
           ELSE
               DO J = 1, M
                   X(J,I) = X(J,I) + Vl(J,K) * V2(I,K)
               END DO
           END IF
       END DO
```

The code example can be rewritten to use the Sun Performance Library routine `DGER`, as shown here.

```
       DO I = 1, N
           IF (V2(I,K) .LT. 0.0) THEN
              V2(I,K) = 0.0
           END IF
       END DO
       CALL DGER (M, N, 1.0D0, X, LDX, Vl(l,K), 1, V2(1,K), 1)
```

The same code example can also be rewritten using Fortran 95 specific statements, as shown here.

```
WHERE (V(1:N,K) .LT. 0.0) THEN
       V(1:N,K) = 0.0
END WHERE
CALL DGER (M, N, 1.0D0, X, LDX, Vl(l,K), 1, V2(1,K), 1)
```

Because the code to replace negative numbers with zero in V2 has no natural analog in Sun Performance Library, that code is pulled out of the outer loop. With that code removed to its own loop, the rest of the loop is a rank- 1 update of the general matrix x that can be replaced with the DGER routine from BLAS.

The amount of performance increase can also depend on the data the Sun Performance Library routine uses. For example, if V2 contains many negative or zero values, the majority of the time might not be spent in the rank- 1 update. In this case, replacing the code with a call to DGER might not increase performance.

Evaluating other loop indexes can affect the Sun Performance Library routine used. For example, if the reference to K is a loop index, the loops in the code sample shown above might be part of a larger code structure, where the loops over DGEMV or DGER could be converted to some form of matrix multiplication. If so, a single call to a matrix multiplication routine can increase performance more than using a loop with calls to DGER.

Because all Sun Performance Library routines are MT-safe (multithread safe), using the auto-parallelizing compiler to parallelize loops that contain calls to Sun Performance Library routines can increase performance on MP platforms.

An example of combining a Sun Performance Library routine with an auto-parallelizing compiler parallelization directive is shown in the following code example.

```
      C$PAR DOALL
      DO I = 1, N
             CALL DGBMV ('No transpose', N, N, ALPHA, A, LDA,
     $     B(l,I), 1, BETA, C(l,I), 1)
      END DO
```

Sun Performance Library contains a routine named DGBMV to multiply a banded matrix by a vector. By putting this routine into a properly constructed loop, use Sun Performance Library routines can be used to multiply a banded matrix by a matrix. The compiler will not parallelize this loop by default, because the presence of subroutine calls in a loop inhibits parallelization. However, Sun Performance Library routines are MT-safe, so a user can use parallelization directives that instruct the compiler to parallelize this loop.

Compiler directives can also be used to parallelize a loop with a subroutine call that ordinarily would not be parallelizable. For example, it is ordinarily not possible to parallelize a loop containing a call to some of the linear system solvers, because some vendors have implemented those routines using code that is not MT-safe. Loops containing calls to the expert drivers of the linear system solvers (routines whose names end in SVX) are usually not parallelizable with other implementations of LAPACK. Because the implementation of LAPACK in Sun Performance Library allows parallelization of loops containing such calls, users of MP platforms can get additional performance by parallelizing these loops.

C Interfaces

The Sun Performance Library routines can be called from within a FORTRAN 77, Fortran 95, or C program. However, C programs must still use the FORTRAN 77 calling sequence.

Sun Performance Library contains native C interfaces for each of the routines contained in LAPACK, BLAS, FFTPACK, VFFTPACK, and LINPACK. The Sun Performance Library C interfaces have the following features:

- Function names have C names
- Function interfaces follow C conventions
- C functions do not contain redundant or unnecessary arguments for a C function

The following example compares the standard LAPACK Fortran interface and the Sun Performance Library C interfaces for the DGBCON routine.

```
CALL DGBCON (NORM, N, NSUB, NSUPER, DA, LDA, IPIVOT, DANORM,
             DRCOND, DWORK, IWORK2, INFO)
void dgbcon(char norm, int n, int nsub, int nsuper, double *da,
            int lda, int *ipivot, double danorm, double drcond,
            int *info)
```

Note that the names of the arguments are the same and that arguments with the same name have the same base type. Scalar arguments that are used only as input values, such as NORM and N, are passed by value in the C version. Arrays and scalars that will be used to return values are passed by reference.

The Sun Performance Library C interfaces improve on CLAPACK, available on Netlib, which is an f2c translation of the standard libraries. For example, all of the CLAPACK routines are followed by a trailing underscore to maintain compatibility with Fortran compilers, which often postfix routine names in the object (.o) file with an underscore. The Sun Performance Library C interfaces do not require a trailing underscore.

Sun Performance Library C interfaces use the following conventions:

- Input-only scalars are passed by value rather than by reference. Complex and double complex arguments are not considered scalars because they are not implemented as a scalar type by C.
- Complex scalars can be passed as either structures or arrays of length 2.
- Types of arguments must match even after C does type conversion. For example, be careful when passing a single precision real value, because a C compiler can automatically promote the argument to double precision.
- Arrays are stored columnwise. For Fortran programmers, this is the natural order in which arrays are stored. For C programmers, this is the transpose of the order in which they usually work. References in the documentation and man pages to rows refer to columns and vice versa.
- Array indices are based at zero in conformance with C conventions, rather than being based at one in conformance with Fortran conventions.

  For example, the Fortran interface to `IDAMAX`, which C programs access as `idamax_`, would return a 1 to indicate the first element in a vector. The C interface to `idamax`, which C programs access as `idamax`, would return a 0 to indicate the first element of a vector. This convention is observed in function return values, permutation vectors, and anywhere else that vector or array indices are used.

Note – Some Sun Performance Library routines use `malloc` internally, so user codes that make calls to Sun Performance Library and to `sbrk` might not work correctly.

Sun Performance Library uses global integer registers `%g2`, `%g3`, and `%g4` in 32-bit mode and `%g2` through `%g5` in 64-bit mode as scratch registers. User code should not use these registers for temporary storage, and then call a Sun Performance Library routine. The data will be overwritten when the Sun Performance Library routine uses these registers.

C Examples

Transforming user-written code sequences into calls to Sun Performance Library routines increases application performance. The following code example adapted from LAPACK shows one example.

```
int    i;
float a[n], b[n], largest;

largest = a[0];
for (i = 0; i < n; i++)
{
if (a[i] > largest)
    largest = a[i];
    if (b[i] > largest
    largest = b[i];
}
```

No Sun Performance Library routine exactly replicates the functionality of this code example. However, the code can be accelerated by replacing it with several calls to the Sun Performance Library routine `isamax`, as shown in the following code example.

```
int    i, large_index;
float a[n], b[n], largest;

large_index = isamax (n, a, l);
largest = a[large_index];
large_index = isamax (n, b, l);
if (b[large_index] > largest)
     largest = b[large_index];
```

Compare the differences between calling the native C `isamax` routine in Sun Performance Library, shown in the previous code example, with calling the `isamax` routine in CLAPACK, shown in the following code example.

```
/* 1. Declare scratch variable to allow 1 to be passed by value */
int one = l;
/* 2. Append underscore to conform to FORTRAN naming system     */
/* 3. Pass all arguments, even scalar input-only, by reference  */
/* 4. Subtract one to convert from FORTRAN indexing conventions */
large_index = isamax_ (&n, a, &one) - l;
largest = a[large_index]; large_index = isamax_ (&n, b, &one) - l;
if (b[large_index] > largest)
     largest = b[large_index];
```

CHAPTER 3

SPARC Optimization and Parallel Processing

This chapter describes how to use compiler and linking options to optimize applications for:

- Specific SPARC instruction set architectures
- 64-bit enabled Solaris operating environment
- Parallel processing

TABLE 3-1 shows a comparison of the 32-bit and 64-bit operating environments. These items are described in greater detail in the following sections.

TABLE 3-1 Comparison of 32-bit and 64-bit Operating Environments

	32-bit (ILP 32)	64-bit (LP64)
`-xarch`	`v8`, `v8plusa`, `v8plusb`	`v9`, `v9a`, `v9b`
Fortran Integers	`INTEGER`, `INTEGER*4`	`INTEGER*8`
C Integers	`int`	`long`
Floating-point	`S/D/C/Z`	`S/D/C/Z`
API	Names of routines	Names of routines with `_64` suffix

Using Sun Performance Library on SPARC Platforms

The Sun Performance Library was compiled using the `f95` compiler provided with this release. The Sun Performance Library routines were compiled using `-dalign` and `-xarch` set to `v8`, `v8plusa`, or `v9a`.

For each `-xarch` option used to compile the libraries, there is a library compiled with `-xparallel` and a library compiled without `-xparallel`. When linking the program, use `-dalign`, `-xlic_lib=sunperf`, and the same command line options that were used when compiling. If `-dalign` cannot be used in the program, supply a trap 6 handler as described in "Getting Started With Sun Performance Library" on page 13. If compiling with a value of `-xarch` that is not one of `[v8|v8plusa|v9a]`, the compiler driver will select the closest match.

Sun Performance Library is linked into an application with the `-xlic_lib` switch rather than the `-l` switch that is used to link in other libraries, as shown here.

```
my_system% f95 -dalign my_file.f -xlic_lib=sunperf
```

Compiling for SPARC Platforms

Applications using Sun Performance Library can be optimized for specific SPARC instruction set architectures and for a 64-bit enabled Solaris operating environment. The optimization for each architecture is targeted at one implementation of that architecture and includes optimizations for other architectures when it does not degrade the performance of the primary target.

Compile with the most appropriate `-xarch=` option for best performance. At link time, use the same `-xarch=` option that was used at compile time to select the version of the Sun Performance Library optimized for a specific SPARC instruction set architecture.

Note – Using SPARC-specific optimization options increases application performance on the selected instruction set architecture, but limits code portability. When using these optimization options, the resulting code can be run only on systems using the specific SPARC chip from Sun Microsystems and, in some cases, a specific Solaris operating environment (32-bit or 64-bit Solaris 7 or Solaris 8).

The SunOS™ command `isalist`(1) can be used to display a list of the native instruction sets executable on a particular platform. The names output by `isalist` are space-separated and are ordered in the sense of best performance.

For a detailed description of the different `-xarch` options, refer to the *Fortran User's Guide* or the *C User's Guide*.

Use the following command line options to compile for 32-bit addressing in a 32-bit enabled Solaris operating environment:

- **UltraSPARC I™ or UltraSPARC II™ systems.** Use `-xarch=v8plus` or `-xarch=v8plusa`.
- **UltraSPARC III™ systems.** Use `-xarch=v8plus` or `-xarch=v8plusb`.

Use the following command line options to compile for 64-bit addressing in a 64-bit enabled Solaris operating environment.

- **UltraSPARC I or UltraSPARC II systems.** Use `-xarch=v9` or `-xarch=v9a`.
- **UltraSPARC III systems.** Use `-xarch=v9` or `-xarch=v9b`.

Compiling Code for a 64-Bit Enabled Solaris Operating Environment

To compile code for a 64-bit enabled Solaris operating environment, use `-xarch=v9[a|b]` and convert all integer arguments to 64-bit arguments. 64-bit routines require the use of 64-bit integers.

Sun Performance Library provides 32-bit and 64-bit interfaces. To use the 64-bit interfaces:

- **Modify the Sun Performance Library routine name.** For C, FORTRAN 77, and Fortran 95 code, append `_64` to the names of Sun Performance Library routines (for example, `rfftf_64` or `CFFTB_64`). For Fortran 95 code with the `USE SUNPERF` statement, the `_64` suffix is not strictly required for specific interfaces, such as `DGEMM`. The `_64` suffix is still required for the generic interfaces, such as `GEMM`.
- **Promote integers to 64 bits.** Double precision variables and the real and imaginary parts of double complex variables are already 64 bits. Only the integers are promoted to 64 bits.

64-Bit Integer Arguments

These additional 64-bit-integer interfaces are available only in the v9, v9a, and v9b libraries. Codes compiled for 32-bit operating environments (`-xarch` set to `v8plusa` or `v8plusb`) can not call the 64-bit-integer interfaces.

To call the 64-bit-integer interfaces directly, append the suffix `_64` to the standard library name. For example, use `daxpy_64()` in place of `daxpy()`.

However, if calling the 64-bit integer interfaces indirectly, do not append `_64` to the name of the Sun Performance Library routine. Calls to the Sun Performance Library routine will access a 32-bit wrapper that promotes the 32-bit integers to 64-bit integers, calls the 64-bit routine, and then demotes the 64-bit integers to 32-bit integers.

For best performance, call the routine directly by appending `_64` to the routine name.

For C programs, use `long` instead of `int` arguments. The following code example shows calling the 64-bit integer interfaces directly.

```
#include <sunperf.h>
long n, incx, incy;
double alpha, *x, *y;
daxpy_64(n, alpha, x, incx, y, incy);
```

The following code example shows calling the 64-bit integer interfaces indirectly.

```
#include <sunperf.h>
int  n, incx, incy;
double alpha, *x, *y;
daxpy   (n, alpha, x, incx, y, incy);
```

For Fortran programs, use 64-bit integers for all integer arguments. The following methods can be used to convert integer arguments to 64-bits:

- To promote all default integers (integers declared without explicit byte sizes) and literal integer constants from 32 bits to 64 bits, compile with `-xtypemap=integer:64`.
- To promote specific integer declarations, change `INTEGER` or `INTEGER*4` to `INTEGER*8`.
- To promote integer literal constants, append `_8` to the constant. This is Fortran 95 style syntax, but it is also recognized by the FORTRAN 77 compiler.

Consider the following code example.

```
INTEGER*8 N
REAL*8 ALPHA, X(N), Y(N)

! _64 SUFFIX: N AND 1_8 ARE 64-BIT INTEGERS
CALL DAXPY_64(N,ALPHA,X,1_8,Y,1_8)
```

INTEGER*8 arguments cannot be used in a 32-bit environment. Routines in the 32-bit libraries, v8, v8plusa, v8plusb, cannot be called with 64-bit arguments. However, the 64-bit routines can be called with 32-bit arguments.

When passing constants in Fortran 95 code that have not been compiled with -xtypemap, append _8 to literal constants to effect the promotion. For example, when using Fortran 95, change CALL DSCAL(20,5.26D0,X,1) to CALL DSCAL(20_8,5.26D0,X,1_8). This example assumes USE SUNPERF is included in the code, because the _64 has not been appended to the routine name.

The following code example shows calling CAXPY from Fortran 95 using 32-bit arguments.

```
       PROGRAM TEST
       COMPLEX ALPHA
       INTEGER INCX, INCY, N
       COMPLEX X(*), Y(*)

       CALL CAXPY(N, ALPHA, X, INCX, Y, INCY)
```

The following code example shows calling CAXPY from Fortran 95 (without the USE SUNPERF statement) using 64-bit arguments.

```
       PROGRAM TEST
       COMPLEX   ALPHA
       INTEGER*8 INCX, INCY, N
       COMPLEX   X(*), Y(*)

       CALL CAXPY_64(N, ALPHA, X, INCX, Y, INCY)
```

When using 64-bit arguments, the _64 must be appended to the routine name if the USE SUNPERF statement is not used.

The following Fortran 95 code example shows calling CAXPY using 64-bit arguments.

```
        PROGRAM TEST
        USE SUNPERF
        .
        .
        .
        COMPLEX   ALPHA
        INTEGER*8 INCX, INCY, N
        COMPLEX   X(*), Y(*)

        CALL CAXPY(N, ALPHA, X, INCX, Y, INCY)
```

In C routines, the size of long is 32 bits when compiling for V8 or V8plus and 64 bits when compiling for V9. The following code example shows calling the dgbcon routine using 32-bit arguments.

```
void dgbcon(char norm, int n, int nsub, int nsuper, double *da,
            int lda, int *ipivot, double danorm, double drcond,
            int *info)
```

The following code example shows calling the dgbcon routine using 64-bit arguments.

```
void dgbcon_64 (char norm, long n, long nsub, long nsuper,
                double *da, long lda, long *ipivot, double danorm,
                double *drcond, long *info)
```

Parallel Processing

If using multithreading, use one of the following options:

- For code compiled with automatic or explicit compiler parallelization, use the same parallelization option (`-xparallel`, `-xexplicitpar`, or `-xautopar`) at link time as at compile time, as shown in the following example.

```
% cc  -dalign -xarch=... -xparallel a.c   -xlic_lib=sunperf
or
% f95 -dalign -xarch=... -xparallel a.f95 -xlic_lib=sunperf
```

- For code that uses POSIX/Solaris threads, use `-mt` on the link line, as shown in the following example.

```
% cc  -dalign -xarch=... -mt        a.c   -xlic_lib=sunperf
or
% f95 -dalign -xarch=... -mt        a.f95 -xlic_lib=sunperf
```

Sun Performance Library does not support mixing compiler parallelization and POSIX/Solaris multithreading.

Note – The Fortran compiler parallelization features require a Forte for HPC license.

Run-time Issues

At run time, if running with compiler parallelization, Sun Performance Library uses the same pool of threads that the compiler does. The per-thread stack size must be set to at least 4 Mbytes with the `STACKSIZE` environment variable, as follows:

```
% setenv STACKSIZE 4000
```

Setting the `STACKSIZE` environment variable is not required for programs running with POSIX/Solaris threads. In this case, Sun Performance Library will create its own threads and ensure that the stack sizes are large enough to accommodate the program's needs.

Degree of Parallelism

Sun Performance Library will attempt to parallelize each Sun Performance Library call according to the user's parallelization model by using either explicit threads or loop-based compiler multithreading.

The number of threads Sun Performance Library routines will attempt to use is set at run time by the user with the `PARALLEL` environment variable. The `PARALLEL` environment variable can be overridden by calls to the Sun Performance Library `USE_THREADS` routine.

For example, if user programs with POSIX/Solaris-thread codes are linked with -mt, each Sun Performance Library call will produce `PARALLEL` threads. The code will oversubscribe the machine if:

- One bound thread per CPU is created
- Each thread makes a Sun Performance Library call
- `PARALLEL` is set to a value greater than one

For codes using compiler parallelization, Sun Performance Library routines are parallelized with loop-based compiler directives. Because nested parallelism is not supported, Sun Performance Library calls made from a parallel region will not be further parallelized.

In the following code example, none of the calls to `DGEMM` is parallelized, because the loop is parallelized and only one level of parallelization is supported.

```
!$<some parallelization directive>
 DO I = 1, N
   CALL DGEMM(...)
 END DO
```

The loop consists of many `DGEMM` instances running in parallel with one another, but each `DGEMM` instance uses only one thread.

In the following code example, the loop is not parallelized.

```
DO I = 1, N
    CALL DGEMM(...)
END DO
```

If the code is linked for parallelization with -mt, `-xparallel`, `-xexplicitpar`, or `-xautopar`, the individual calls to `DGEMM` will be parallelized. The number of threads used by each `DGEMM` call will be taken from the run-time value of the environment variable `PARALLEL`. However, if a higher-level loop has already parallelized this region, no further parallelization would be performed.

The number of OpenMP threads can be set by a variety of means. For example, by setting the PARALLEL or the OMP_NUM_THREADS environment variable or by setting the OMP_SET_NUM_THREADS() run-time call. If both environment variables are set, they must be set to the same value. If the run-time function is called, it overrides any environment variable setting.

The degree of parallelization within a pure-OpenMP code can be set by either the PARALLEL or the OMP_NUM_THREADS environment variable. The Sun Performance Library USE_THREADS() routine can also be used to set the degree of parallelism for Sun Performance Library calls, which overrides the PARALLEL value.

In the following code example, each DGEMM call would be parallelized.

```
!$PAR DOSERIAL*
 DO I = 1, N
   CALL DGEMM(...)
 END DO
```

Note that the DOSERIAL* directive suppresses parallelization, but only for the loop nest within the same subroutine and it is overridden by any other directive within that nest. The DOSERIAL* directive does not impact parallelization within Sun Performance Library.

In the following code example, there will be at most 2-way parallelism, regardless of the setting of PARALLEL or of the number of OpenMP threads.

```
!$OMP PARALLEL SECTIONS
 !$OMP SECTION
 DO I = 1, N / 2
   CALL DGEMM(...)
 END DO
 !$OMP SECTION
 DO I = N / 2 + 1, N
   CALL DGEMM(...)
 END DO
 !$OMP END PARALLEL SECTIONS
```

Only one level of parallelism exists, which are the two sections. Further parallelism within a DGEMM() call is suppressed.

Synchronization Mechanisms

The underlying parallelization model determines the Sun Performance Library behavior.

The two basic modes of multithreading, compiler parallelization and POSIX/Solaris threads, use two different types of synchronization mechanisms. Compiler parallelized code uses spin waits, which produce the most responsive synchronization operations, but aggressively consume CPU cycles. Compiler parallelized code produces optimal performance when each thread has a dedicated CPU, but wastes resources when other jobs or threads are also competing for CPUs.

However, codes that explicitly use POSIX/Solaris threads use synchronization functions from `libthread`. These synchronization functions are less responsive, but they relinquish the CPU when the thread is idle, providing good throughput and resource usage in a shared (oversubscribed) environment.

With compiler parallelization, the environment variable `SUNW_MP_THR_IDLE` can be used at run time to alter the spin-wait characteristics of the threads. Legal settings of `SUNW_MP_THR_IDLE` are as follows.

```
% setenv SUNW_MP_THR_IDLE spin
% setenv SUNW_MP_THR_IDLE 2s
% setenv SUNW_MP_THR_IDLE 100ms
```

These settings would cause threads to spin wait (default behavior), spin for 2 seconds before sleeping, or spin for 100 milliseconds before sleeping, respectively.

The link-time option `-xlic_lib=sunperf` links in Sun Performance Library functions that employ the same parallelization model as the user code, as indicated by `-mt` or by a compiler-parallelization option (`-xparallel`, `-xexplicitpar`, or `-xautopar`). Using Sun Performance Library routines do not change the spin-wait behavior of the code.

Parallel Processing Examples

The following sections demonstrate using the PARALLEL environment variable and the compile and linking options for creating code that supports using:

- A single processor
- Multiple processors

Using a Single Processor

To use a single processor:

1. **Call one or more of the routines.**
2. **Link with `-xlic_lib=sunperf` specified at the end of the command line.**

   Do not compile or link with `-xparallel`, `-xexplicitpar`, or `-xautopar`.
3. **Make sure the `PARALLEL` environment variable is unset or set equal to 1.**

   To following example shows how to compile and link with `libsunperf.so`.

```
cc -dalign -xarch=... any.c -xlic_lib=sunperf
or
f95 -dalign -xarch=... any.f95 -xlic_lib=sunperf
```

Using Multiple Processors

To compile for multiple processors:

- Use the same parallelization option for the compiling and linking commands.
- Specify the number of processors at runtime with the PARALLEL environment variable before running the executable.

For example, to use 24 processors, type the following commands.

```
my_system% f95 -dalign -mt my_app.f -xlic_lib=sunperf
my_system% setenv PARALLEL 24
my_system% ./a.out
```

The previous example allows Sun Performance Library routines to run in parallel, but no part of the user code `my_app.f` will run in parallel. For the compiler to attempt to parallelize `my_app.f`, either `-xparallel` or `-explicitpar` is required on the compile line.

Note – Parallel processing options require using either the `-dalign` command-line option or establishing a trap 6 handler, as described in “Enabling Trap 6” on page 14. When using C, do not use `-misalign`.

To use multiple processors:

1. **Call one or more of the routines.**
2. **Link with `-xlic_lib=sunperf` specified at the end of the command line.**

   Compile and link with `-xparallel`, `-xexplicitpar`, or `-xautopar.`
3. **Set `PARALLEL` to the number of available processors.**

   To following example shows how to compile and link with `libsunperf_mt.so`.

```
cc -dalign -xarch=... -xparallel any.c -xlic_lib=sunperf
or
f95 -dalign -xarch=... -xparallel any.f95 -xlic_lib=sunperf
```

FFT Example

FFT and VFFT routines have been modified to take advantage of parallelization enhancements. For example, FFT and VFFT routines can be used in parallelized loops, as shown here.

```
      CALL CFFTI (M, WSAVE)
C$PAR DOALL SHARED(M, WSAVE, N, C), PRIVATE(I)
      DO I = 1, N
        CALL CFFTF (M, C(1, I), WSAVE)
        CALL CFFTB (M, C(1, I), WSAVE)
      END DO
```

CHAPTER 4

Working With Matrices

Most matrices can be stored in ways that save both storage space and computation time. Sun Performance Library uses the following storage schemes:

- Banded storage
- Packed storage

The Sun Performance Library processes matrices that are in one of four forms:

- General
- Triangular
- Symmetric
- Tridiagonal

Storage schemes and matrix types are described in the following sections.

Matrix Storage Schemes

Some Sun Performance Library routines that work with arrays stored normally have corresponding routines that take advantage of these special storage forms. For example, `DGBMV` will form the product of a general matrix in banded storage and a vector, and `DTPMV` will form the product of a triangular matrix in packed storage and a vector.

Banded Storage

A banded matrix is stored so the *j*th column of the matrix corresponds to the *j*th column of the Fortran array.

The following code copies a banded general matrix in a general array into banded storage mode.

```
C     Copy the matrix A from the array AG to the array AB. The
C     matrix is stored in general storage mode in AG and it will
C     be stored in banded storage mode in AB. The code to copy
C     from general to banded storage mode is taken from the
C     comment block in the original DGBFA by Cleve Moler.
C
      NSUB = 1
      NSUPER = 2
      NDIAG = NSUB + 1 + NSUPER
      DO ICOL = 1, N
        I1 = MAX0 (1, ICOL - NSUPER)
        I2 = MIN0 (N, ICOL + NSUB)
        DO IROW = I1, I2
          IROWB = IROW - ICOL + NDIAG
          AB(IROWB,ICOL) = AG(IROW,ICOL)
        END DO
      END DO
```

Note that this method of storing banded matrices is compatible with the storage method used by LAPACK, BLAS, and LINPACK, but is inconsistent with the method used by EISPACK.

Packed Storage

A packed vector is an alternate representation for a triangular, symmetric, or Hermitian matrix. An array is packed into a vector by storing the elements sequentially column by column into the vector. Space for the diagonal elements is always reserved, even if the values of the diagonal elements are known, such as in a unit diagonal matrix.

An upper triangular matrix or a symmetric matrix whose upper triangle is stored in general storage in the array A, can be transferred to packed storage in the array AP as shown below. This code comes from the comment block of the LAPACK routine DTPTRI.

```
    JC = 1
    DO J = 1, N
       DO I = 1, J
          AP(JC+I-1) = A(I,J)
       END DO
       JC = JC + J
     END DO
```

Similarly, a lower triangular matrix or a symmetric matrix whose lower triangle is stored in general storage in the array A, can be transferred to packed storage in the array AP as shown below:

```
    JC = 1
    DO J = 1, N
       DO I = J, N
          AP(JC+I-1) = A(I,J)
       END DO
       JC = JC + N - J + 1
    END DO
```

Matrix Types

The general matrix form is the most common matrix, and most operations performed by the Sun Performance Library can be done on general arrays. In many cases, there are routines that will work with the other forms of the arrays. For example, DGEMM will form the product of two general matrices and DTRMM will form the product of a triangular and a general matrix.

General Matrices

A general matrix is stored so that there is a one-to-one correspondence between the elements of the matrix and the elements of the array. Element A_{ij} of a matrix A is stored in element `A(I,J)` of the corresponding array A. The general form is the most common form. A general matrix, because it is dense, has no special storage scheme. In a general banded matrix, however, the diagonal of the matrix is stored in the row below the upper diagonals.

For example, as shown below, the general banded matrix can be represented with banded storage. Elements shown with the symbol × are never accessed by routines that process banded arrays.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

General Banded Matrix

$$\begin{bmatrix} \times & \times & a_{13} & a_{24} & a_{35} \\ \times & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & \times \end{bmatrix}$$

General Banded Array in Banded Storage

Triangular Matrices

A triangular matrix is stored so that there is a one-to-one correspondence between the nonzero elements of the matrix and the elements of the array, but the elements of the array corresponding to the zero elements of the matrix are never accessed by routines that process triangular arrays.

A triangular matrix can be stored using packed storage.

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Triangular Matrix

$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{22} \\ a_{32} \\ a_{33} \end{bmatrix}$$

Triangular Array in Packed Storage

A triangular banded matrix can be stored using banded storage as shown below. Elements shown with the symbol × are never accessed by routines that process banded arrays.

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ 0 & a_{32} & a_{33} \end{bmatrix}$$

Triangular Banded Matrix

$$\begin{bmatrix} a_{11} & a_{22} & a_{33} \\ a_{21} & a_{32} & \times \end{bmatrix}$$

Triangular Banded Array
in Banded Storage

Symmetric Matrices

A symmetric matrix is similar to a triangular matrix in that the data in either the upper or lower triangle corresponds to the elements of the array. The contents of the other elements in the array are assumed and those array elements are never accessed by routines that process symmetric or Hermitian arrays.

A symmetric matrix can be stored using packed storage.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Symmetric Matrix

$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{22} \\ a_{32} \\ a_{33} \end{bmatrix}$$

Symmetric Array in Packed Storage

A symmetric banded matrix can be stored using banded storage as shown below. Elements shown with the symbol $\times$ are never accessed by routines that process banded arrays.

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix} \qquad \begin{bmatrix} \times & a_{12} & a_{23} & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \\ a_{21} & a_{32} & a_{43} & \times \end{bmatrix}$$

Symmetric Banded Matrix

Symmetric Banded Array in Banded Storage

Tridiagonal Matrices

A tridiagonal matrix has elements only on the main diagonal, the first superdiagonal, and the first subdiagonal. It is stored using three 1-dimensional arrays.

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix} \qquad \begin{bmatrix} a_{21} \\ a_{32} \\ a_{43} \end{bmatrix} \quad \begin{bmatrix} a_{11} \\ a_{22} \\ a_{33} \\ a_{44} \end{bmatrix} \quad \begin{bmatrix} a_{12} \\ a_{23} \\ a_{34} \end{bmatrix}$$

Tridiagonal Matrix

Tridiagonal Array in Tridiagonal Storage

Sparse Matrices

The Sun Performance Library sparse solver package is a collection of routines that efficiently factor and solve sparse linear systems of equations. Use the sparse solver package to:

- Solve symmetric, structurally symmetric, and unsymmetric coefficient matrices
- Specify a choice of ordering methods, including user-specified orderings

The sparse solver package contains interfaces for FORTRAN 77. Fortran 95 and C interfaces are not currently provided. To use the sparse solver routines from Fortran 95, use the FORTRAN 77 interfaces. To use the sparse solver routines with C, append an underscore to the routine name (`dgssin_()`, `dgssor_()`, and so on), pass arguments by reference, and use 1-based array indexing.

Sparse Solver Matrix Data Formats

Sparse matrices are usually represented in formats that minimize storage requirements. By taking advantage of the sparsity and not storing zeros, considerable storage space can be saved. The storage format used by the general sparse solver is the compressed sparse column (CSC) format (also called the Harwell-Boeing format).

The CSC format represents a sparse matrix with two integer arrays and one floating point array. The integer arrays (colptr and rowind) specify the location of the nonzeros of the sparse matrix, and the floating point array (values) is used for the nonzero values.

The column pointer (colptr) array consists of n+1 elements where colptr(i) points to the beginning of the ith column, and colptr(i + 1) – 1 points to the end of the ith column. The row indices (rowind) array contains the row indices of the nonzero values. The values arrays contains the corresponding nonzero numerical values.

The following matrix data formats exist for a sparse matrix of *neqns* equations and *nnz* nonzeros:

- Symmetric
- Structurally symmetric
- Unsymmetric

The most efficient data representation often depends on the specific problem. The following sections show examples of sparse matrix data formats.

Symmetric Sparse Matrices

A symmetric sparse matrix is a matrix where a(i, j) = a(j, i) for all i and j. Because of this symmetry, only the lower triangular values need to be passed to the solver routines. The upper triangle can be determined from the lower triangle.

An example of a symmetric matrix is shown below. This example is derived from A. George and J. W-H. Liu. "Computer Solution of Large Sparse Positive Definite Systems."

$$A = \begin{bmatrix} 4.0 & 1.0 & 2.0 & 0.5 & 2.0 \\ 1.0 & 0.5 & 0.0 & 0.0 & 0.0 \\ 2.0 & 0.0 & 3.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & 0.0 & 0.625 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 16.0 \end{bmatrix}$$

To represent A in CSC format:

- colptr: 1, 6, 7, 8, 9, 10
- rowind: 1, 2, 3, 4, 5, 2, 3, 4, 5
- values: 4.0, 1.0, 2.0, 0.5, 2.0, 0.5, 3.0, 0.625, 16.0

Structurally Symmetric Sparse Matrices

A structurally symmetric sparse matrix has nonzero values with the property that if a(i, j) $\neq$ 0, then a(j, i) $\neq$ 0 for all i and j. When solving a structurally symmetric system, the entire matrix must be passed to the solver routines.

An example of a structurally symmetric matrix is shown below.

$$A = \begin{bmatrix} 1.0 & 3.0 & 0.0 & 0.0 \\ 2.0 & 4.0 & 0.0 & 7.0 \\ 0.0 & 0.0 & 6.0 & 0.0 \\ 0.0 & 5.0 & 0.0 & 8.0 \end{bmatrix}$$

To represent A in CSC format:

- colptr: 1, 3, 6, 7, 9
- rowind: 1, 2, 1, 2, 4, 3, 2, 4
- values: 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0

Unsymmetric Sparse Matrices

An unsymmetric sparse matrix does not have a(*i*, *j*) = a(*j*, *i*) for all *i* and *j*. The structure of the matrix does not have an apparent pattern. When solving an unsymmetric system, the entire matrix must be passed to the solver routines. An example of an unsymmetric matrix is shown below.

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 6.0 & 0.0 & 0.0 & 9.0 \\ 3.0 & 0.0 & 7.0 & 0.0 & 0.0 \\ 4.0 & 0.0 & 0.0 & 8.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 0.0 & 10.0 \end{bmatrix}$$

To represent A in CSC format:

- colptr: 1, 6, 7, 8, 9, 11
- rowind: 1, 2, 3, 4, 5, 2, 3, 4, 2, 5
- values: 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0

Sun Performance Library Sparse BLAS

The Sun Performance Library sparse BLAS package is based on the following two packages:

- Netlib Sparse BLAS package, by Dodson, Grimes, and Lewis consists of sparse extensions to the Basic Linear Algebra Subroutines that operate on sparse vectors.
- NIST (National Institute of Standards and Technology) Fortran Sparse BLAS Library consists of routines that perform matrix products and solution of triangular systems for sparse matrices in a variety of storage formats.

Refer to the following sources for additional sparse BLAS information.

- For information on the Sun Performance Library Sparse BLAS routines, refer to the section 3P man pages for the individual routines.
- For more information on the Netlib Sparse BLAS package refer to `http://www.netlib.org/sparse-blas/index.html`.
- For more information on the NIST Fortran Sparse BLAS routines, refer to `http://math.nist.gov/spblas/`

Naming Conventions

The Netlib Sparse BLAS and NIST Fortran Sparse BLAS Library routines each use their own naming conventions, as described in the following two sections.

Netlib Sparse BLAS

Each Netlib Sparse BLAS routine has a name of the form Prefix-Root-Suffix where the:

- Prefix represents the data type.
- Root represents the operation.
- Suffix represents whether or not the routine is a direct extension of an existing dense BLAS routine.

TABLE 4-1 lists the naming conventions for the Netlib Sparse BLAS vector routines.

TABLE 4-1 Netlib Sparse BLAS Naming Conventions

Operation	Root of Name	Prefix and Suffix
Dot product	`-DOT-`	`S-I D-I C-UI Z-UI C-CI Z-CI`
Scalar times a vector added to a vector	`-AXPY-`	`S-I D-I C-I Z-I`
Apply Givens rotation	`-ROT-`	`S-I D-I`
Gather x into y	`-GTHR-`	`S- D- C- Z- S-Z D-Z C-Z Z-Z`
Scatter x into y	`-SCTR-`	`S- D- C- Z-`

The prefix can be one of the following data types:

- S: `SINGLE`
- D: `DOUBLE`
- C: `COMPLEX`
- Z: `COMPLEX*16` or `DOUBLE COMPLEX`

The `I`, `CI`, and `UI` suffixes denote sparse BLAS routines that are direct extensions to dense BLAS routines.

NIST Fortran Sparse BLAS

Each NIST Fortran Sparse BLAS routine has a six-character name of the form *XYYYZZ* where:

- *X* represents the data type.
- *YYY* represents the sparse storage format.
- *ZZ* represents the operation.

TABLE 4-2 shows the values for *X*, *Y*, and *Z*.

TABLE 4-2 NIST Fortran Sparse BLAS Routine Naming Conventions

X: Data Type		
X	S: single precision D: double precision	
YYY: Sparse Storage Format		
YYY	Single entry formats:	COO: coordinate CSC: compressed sparse column CSR: compressed sparse row DIA: diagonal ELL: ellpack JAD: jagged diagonal SKY: skyline
	Block entry formats:	BCO: block coordinate BSC: block compressed sparse column BSR: block compressed sparse row BDI: block diagonal BEL: block ellpack VBR: block compressed sparse row
ZZ: Operation		
ZZ	MM: matrix-matrix product SM: solution of triangular system (supported for all formats except COO) RP: right permutation (for JAD format only)	

Sparse Solver Routines

The Sun Performance Library sparse solver package contains the routines listed in TABLE 4-3.

TABLE 4-3 Sparse Solver Routines

Routine	Function
DGSSFS()	One call interface to sparse solver
DGSSIN()	Sparse solver initialization
DGSSOR()	Fill reducing ordering and symbolic factorization
DGSSFA()	Matrix value input and numeric factorization
DGSSSL()	Triangular solve
Utility Routine	**Function**
DGSSUO()	Sets user-specified ordering permutation.
DGSSRP()	Returns permutation used by solver.
DGSSCO()	Returns condition number estimate of coefficient matrix.
DGSSDA()	De-allocates sparse solver.
DGSSPS()	Prints solver statistics.

Use the regular interface to solve multiple matrices with the same structure, but different numerical values, as shown below:

```
call dgssin() ! {initialization, input coefficient matrix
              !  structure}
call dgssor() ! {fill-reducing ordering, symbolic factorization}
do m = 1, number_of_structurally_identical_matrices
    call dgssfa() ! {input coefficient matrix values, numeric
                  ! factorization}
    do r = 1, number_of_right_hand_sides
        call dgsssl() ! {triangular solve}
    enddo
enddo
```

The one-call interface is not as flexible as the regular interface, but it covers the most common case of factoring a single matrix and solving some number right-hand sides. Additional calls to `dgsssl()` are allowed to solve for additional right-hand sides, as shown below.

```
call dgssfs() ! {initialization, input coefficient matrix
              ! structure}
              ! {fill-reducing ordering, symbolic factorization}
              ! {input coefficient matrix values, numeric
              ! factorization}
              ! {triangular solve}
do r = 1, number_of_right_hand_sides
    call dgsssl() ! {triangular solve}
enddo
```

Routine Calling Order

To solve problems with the sparse solver package, use the sparse solver routines in the order shown in TABLE 4-4.

TABLE 4-4 Sparse Solver Routine Calling Order

One Call Interface: For solving single matrix	
Start	
`DGSSFS()`	Initialize, order, factor, solve
`DGSSSL()`	Additional solves (optional): repeat `dgsssl()` as needed
`DGSSDA()`	Deallocate working storage
Finish	
End of One-Call Interface	

TABLE 4-4 Sparse Solver Routine Calling Order *(Continued)*

Regular Interface: For solving multiple matrices with the same structure	
Start	
`DGSSIN()`	Initialize
`DGSSOR()`	Order
`DGSSFA()`	Factor
`DGSSSL()`	Solve: repeat `dgssfa()` or `dgsssl()` as needed
`DGSSDA()`	Deallocate working storage
Finish	
End of Regular Interface	

Sparse Solver Examples

CODE EXAMPLE 4-1 shows solving a symmetric system using the one-call interface, and CODE EXAMPLE 4-2 on page 55 shows solving a symmetric system using the regular interface.

CODE EXAMPLE 4-1 Solving a Symmetric System–One-Call Interface

```
my_system% cat example_1call.f
      program example_1call
c
c  This program is an example driver that calls the sparse solver.
c    It factors and solves a symmetric system, by calling the
c    one-call interface.
c
      implicit none

      integer           neqns, ier, msglvl, outunt, ldrhs, nrhs
      character         mtxtyp*2, pivot*1, ordmthd*3
      double precision  handle(150)
      integer           colstr(6), rowind(9)
      double precision  values(9), rhs(5), xexpct(5)
      integer           i
c
c  Sparse matrix structure and value arrays.  From George and Liu,
c  page 3.
```

CODE EXAMPLE 4-1 Solving a Symmetric System–One-Call Interface *(Continued)*

```
c    Ax = b, (solve for x) where:
c
c      4.0   1.0   2.0   0.5   2.0       2.0       7.0
c      1.0   0.5   0.0   0.0   0.0       2.0       3.0
c  A = 2.0   0.0   3.0   0.0   0.0   x = 1.0   b = 7.0
c      0.5   0.0   0.0   0.625 0.0      -8.0      -4.0
c      2.0   0.0   0.0   0.0  16.0      -0.5      -4.0
c
      data colstr / 1, 6, 7, 8, 9, 10 /
      data rowind / 1, 2, 3, 4, 5, 2, 3, 4, 5 /
     data values / 4.0d0, 1.0d0, 2.0d0, 0.5d0, 2.0d0, 0.5d0, 3.0d0,
     &              0.625d0, 16.0d0 /
      data rhs    / 7.0d0, 3.0d0, 7.0d0, -4.0d0, -4.0d0 /
      data xexpct / 2.0d0, 2.0d0, 1.0d0, -8.0d0, -0.5d0 /
c
c  set calling parameters
c
      mtxtyp= 'ss'
      pivot = 'n'
      neqns  = 5
      nrhs   = 1
      ldrhs  = 5
      outunt = 6
      msglvl = 0
      ordmthd = 'mmd'
c
c  call single call interface
c
      call dgssfs ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              values, nrhs  , rhs,    ldrhs , ordmthd,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
```

CODE EXAMPLE 4-1 Solving a Symmetric System–One-Call Interface *(Continued)*

```
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

  300 format(i5,3d20.12) ! i/sol/xexpct values

  400 format(a60,i20) ! fail message, sparse solver error number

      end
my_system% f95 -dalign example_1call.f -xlic_lib=sunperf
my_sytem% a.out
    i             rhs(i)     expected rhs(i)               error
    1  0.200000000000D+01  0.200000000000D+01 -0.528466159722D-13
    2  0.200000000000D+01  0.200000000000D+01  0.105249142734D-12
    3  0.100000000000D+01  0.100000000000D+01  0.350830475782D-13
    4 -0.800000000000D+01 -0.800000000000D+01  0.426325641456D-13
    5 -0.500000000000D+00 -0.500000000000D+00  0.660582699652D-14
```

CODE EXAMPLE 4-2 Solving a Symmetric System–Regular Interface

```
my_system% cat example_ss.f
      program example_ss
c
c  This program is an example driver that calls the sparse solver.
c  It factors and solves a symmetric system.

      implicit none

      integer            neqns, ier, msglvl, outunt, ldrhs, nrhs
      character          mtxtyp*2, pivot*1, ordmthd*3
      double precision   handle(150)
      integer            colstr(6), rowind(9)
      double precision   values(9), rhs(5), xexpct(5)
      integer            i
c
c  Sparse matrix structure and value arrays.  From George and Liu,
c  page 3.
c    Ax = b, (solve for x) where:
c
c      4.0   1.0   2.0   0.5   2.0       2.0       7.0
c      1.0   0.5   0.0   0.0   0.0       2.0       3.0
c  A = 2.0   0.0   3.0   0.0   0.0   x = 1.0   b = 7.0
c      0.5   0.0   0.0   0.625 0.0      -8.0      -4.0
c      2.0   0.0   0.0   0.0  16.0      -0.5      -4.0
c
      data colstr / 1, 6, 7, 8, 9, 10 /
      data rowind / 1, 2, 3, 4, 5, 2, 3, 4, 5 /
      data values / 4.0d0, 1.0d0, 2.0d0, 0.5d0, 2.0d0, 0.5d0,
     &              3.0d0, 0.625d0, 16.0d0 /
      data rhs    / 7.0d0, 3.0d0, 7.0d0, -4.0d0, -4.0d0 /
      data xexpct / 2.0d0, 2.0d0, 1.0d0, -8.0d0, -0.5d0 /
c
c  initialize solver
c
      mtxtyp= 'ss'
      pivot = 'n'
      neqns  = 5
      outunt = 6
      msglvl = 0
```

CODE EXAMPLE 4-2 Solving a Symmetric System–Regular Interface *(Continued)*

```
c
c  call regular interface
c
      call dgssin ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
c
c  ordering and symbolic factorization
c
      ordmthd = 'mmd'
      call dgssor ( ordmthd, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  numeric factorization
c
      call dgssfa ( neqns, colstr, rowind, values, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  solution
c
      nrhs   = 1
      ldrhs  = 5
      call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
```

CODE EXAMPLE 4-2 Solving a Symmetric System–Regular Interface *(Continued)*

```
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

  300 format(i5,3d20.12) ! i/sol/xexpct values

  400 format(a60,i20) ! fail message, sparse solver error number

      end
my_system% f95 -dalign example_ss.f -xlic_lib=sunperf
my_sytem% a.out
    i              rhs(i)     expected rhs(i)               error
    1  0.200000000000D+01  0.200000000000D+01 -0.528466159722D-13
    2  0.200000000000D+01  0.200000000000D+01  0.105249142734D-12
    3  0.100000000000D+01  0.100000000000D+01  0.350830475782D-13
    4 -0.800000000000D+01 -0.800000000000D+01  0.426325641456D-13
    5 -0.500000000000D+00 -0.500000000000D+00  0.660582699652D-14
```

CODE EXAMPLE 4-3 Solving a Structurally Symmetric System With Unsymmetric Values–Regular Interface

```
my_system% cat example_su.f
      program example_su
c
c  This program is an example driver that calls the sparse solver.
c    It factors and solves a structurally symmetric system
c    (w/unsymmetric values).
c
      implicit none

      integer            neqns, ier, msglvl, outunt, ldrhs, nrhs
      character          mtxtyp*2, pivot*1, ordmthd*3
      double precision   handle(150)
      integer            colstr(5), rowind(8)
      double precision   values(8), rhs(4), xexpct(4)
      integer            i
c
c  Sparse matrix structure and value arrays.  Coefficient matrix
c    has a symmetric structure and unsymmetric values.
c    Ax = b, (solve for x) where:
c
c      1.0   3.0   0.0   0.0       1.0        7.0
c      2.0   4.0   0.0   7.0       2.0       38.0
c  A = 0.0   0.0   6.0   0.0   x = 3.0   b = 18.0
c      0.0   5.0   0.0   8.0       4.0       42.0
c
      data colstr / 1, 3, 6, 7, 9 /
      data rowind / 1, 2, 1, 2, 4, 3, 2, 4 /
     data values / 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0, 6.0d0, 7.0d0,
     &              8.0d0 /
      data rhs    / 7.0d0, 38.0d0, 18.0d0, 42.0d0 /
      data xexpct / 1.0d0, 2.0d0, 3.0d0, 4.0d0 /
c
c  initialize solver
c
      mtxtyp= 'su'
      pivot = 'n'
      neqns  = 4
      outunt = 6
      msglvl = 0
```

CODE EXAMPLE 4-3 Solving a Structurally Symmetric System With Unsymmetric Values–Regular Interface *(Continued)*

```
c
c  call regular interface
c
      call dgssin ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
c
c  ordering and symbolic factorization
c
      ordmthd = 'mmd'
      call dgssor ( ordmthd, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  numeric factorization
c
      call dgssfa ( neqns, colstr, rowind, values, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  solution
c
      nrhs   = 1
      ldrhs  = 4
      call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
```

CODE EXAMPLE 4-3 Solving a Structurally Symmetric System With Unsymmetric Values–Regular Interface *(Continued)*

```
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

  300 format(i5,3d20.12)     ! i/sol/xexpct values

  400 format(a60,i20)   ! fail message, sparse solver error number

      end
my_system% f95 -dalign example_su.f -xlic_lib=sunperf
my_system% a.out
    i              rhs(i)     expected rhs(i)               error
    1  0.100000000000D+01  0.100000000000D+01  0.000000000000D+00
    2  0.200000000000D+01  0.200000000000D+01  0.000000000000D+00
    3  0.300000000000D+01  0.300000000000D+01  0.000000000000D+00
    4  0.400000000000D+01  0.400000000000D+01  0.000000000000D+00
```

CODE EXAMPLE 4-4 Solving an Unsymmetric System–Regular Interface

```
my_system% cat example_uu.f
      program example_uu
c
c  This program is an example driver that calls the sparse solver.
c    It factors and solves an unsymmetric system.
c
      implicit none

      integer           neqns, ier, msglvl, outunt, ldrhs, nrhs
      character         mtxtyp*2, pivot*1, ordmthd*3
      double precision  handle(150)
      integer           colstr(6), rowind(10)
      double precision  values(10), rhs(5), xexpct(5)
      integer           i
c
c  Sparse matrix structure and value arrays.  Unsummetric matrix A.
c    Ax = b, (solve for x) where:
c
c       1.0   0.0   0.0   0.0   0.0       1.0        1.0
c       2.0   6.0   0.0   0.0   9.0       2.0       59.0
c   A = 3.0   0.0   7.0   0.0   0.0   x = 3.0   b = 24.0
c       4.0   0.0   0.0   8.0   0.0       4.0       36.0
c       5.0   0.0   0.0   0.0  10.0       5.0       55.0
c
      data colstr / 1, 6, 7, 8, 9, 11 /
      data rowind / 1, 2, 3, 4, 5, 2, 3, 4, 2, 5 /
     data values / 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0, 6.0d0, 7.0d0,
     &              8.0d0, 9.0d0, 10.0d0 /
      data rhs    / 1.0d0, 59.0d0, 24.0d0, 36.0d0, 55.0d0 /
      data xexpct / 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0 /
c
c  initialize solver
c
      mtxtyp= 'uu'
      pivot = 'n'
      neqns  = 5
      outunt = 6
      msglvl = 3
      call dgssin ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
```

CODE EXAMPLE 4-4 Solving an Unsymmetric System–Regular Interface *(Continued)*

```
c
c  ordering and symbolic factorization
c
      ordmthd = 'mmd'
      call dgssor ( ordmthd, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  numeric factorization
c
      call dgssfa ( neqns, colstr, rowind, values, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  solution
c
      nrhs   = 1
      ldrhs  = 5
      call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
```

CODE EXAMPLE 4-4 Solving an Unsymmetric System–Regular Interface *(Continued)*

```
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

  300 format(i5,3d20.12)     ! i/sol/xexpct values

  400 format(a60,i20)    ! fail message, sparse solver error number

      end
my_system% f95 -dalign example_uu.f -xlic_lib=sunperf
my_system% a.out
  i              rhs(i)     expected rhs(i)               error
    1  0.100000000000D+01  0.100000000000D+01  0.000000000000D+00
    2  0.200000000000D+01  0.200000000000D+01  0.000000000000D+00
    3  0.300000000000D+01  0.300000000000D+01  0.000000000000D+00
    4  0.400000000000D+01  0.400000000000D+01  0.000000000000D+00
    5  0.500000000000D+01  0.500000000000D+01  0.000000000000D+00
```

References

The following books and papers provide additional information for the sparse BLAS and sparse solver routines.

- Dodson, D.S, R.G. Grimes, and J.G. Lewis. “Sparse Extensions to the Fortran Basic Linear Algebra Subprograms.” ACM Transactions on Mathematical Software, June 1991, Vol 17, No. 2.
- A. George and J. W-H. Liu. “Computer Solution of Large Sparse Positive Definite Systems.” Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.
- E. Ng and B. W. Peyton. “Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers.” SIAM M. Sci Comput., 14:1034-1056, 1993.
- Ian S. Duff, Roger G. Grimes and John G. Lewis, “User’s Guide for the Harwell-Boeing Sparse Matrix Collection (Release I),” Technical Report TR/PA/92/86, CERFACS, Lyon, France, October 1992.

CHAPTER 5

Using Sun Performance Library Fast Fourier Transform Routines

Many problems involve computing the discrete Fourier transform (DFT) of a periodic sequence of length *N*, where *N* is the number of data points or samples. The number of calculations required to compute the DFT is proportional to N^2. The fast Fourier transform (FFT) was developed to efficiently compute the DFT, where the number of calculations required to compute the FFT is proportional to $N\log_2 N$.

Sun Performance Library™ provides routines for computing the FFT or inverse transform (synthesis) of a sequence of length *N*. The FFT routines are based on FFTPACK and VFFTPACK, which are collections of public domain subroutines available from Netlib (`http://www.netlib.org`). These routines have been enhanced and optimized for SPARC™ platforms, and then bundled with the Sun Performance Library. The Sun Performance Library also includes two-dimensional FFT routines, three-dimensional FFT routines, and convolution and correlation routines.

This chapter describes how to use the Sun Performance Library FFT routines and provides examples of their use. This chapter does not describe the details of the FFT algorithms or the mathematics of the DFT. For more information on these topics, see the sources listed in "References" on page 134.

For information on the Fortran and C interfaces and types of arguments used with each FFT routine, see the section 3P man pages for the individual routines. For example, to display the man page for the `RFFTI` routine, type **`man -s 3P rffti`**. The man page routine names use lowercase letters.

Introduction to the FFTPACK and VFFTPACK Packages

Sun Performance Library contains FFT routines based on FFTPACK and VFFTPACK. Sun Performance Library also contains two-dimensional and three-dimensional FFT routines, which are not a part of FFTPACK or VFFTPACK.

FFTPACK routines operate on a single data sequence of length N. The sequence is stored in a one-dimensional array from which the fast sine, fast cosine, fast Fourier transform, or inverse transform can be computed.

VFFTPACK routines are extensions of FFTPACK routines that operate on two or more data sequences simultaneously. The sequences are stored in a two-dimensional array and are processed individually.

VFFTPACK routines store multiple data sequences in a two-dimensional array, but they compute a linear Fourier transform in only one direction. That is, a one-dimensional Fourier transform is computed for each sequence. The two- and three-dimensional FFT routines in the Sun Performance Library differ from the VFFTPACK routines in that they compute the FFT in more than one direction. The two-dimensional FFT routines compute the FFT on the rows and columns of the input data stored in a two-dimensional array. The three-dimensional FFT routines perform a three-dimensional transform of input data stored in a three-dimensional array.

TABLE 5-1 summarizes some of the similarities and differences between the single vector FFTPACK, multiple vector VFFTPACK routines, two-dimensional FFT routines, and three-dimensional FFT routines.

TABLE 5-1 Comparison Between Single Vector and Multiple Vector Routines

	Single Vector	**Multiple Vector**
One-Dimensional Routines		
Input	Vector of length *N*	An array of vectors
Output	Single transform or inverse transform	Multiple transforms or inverse transform (one transform or inverse transform per sequence)
Results	Unnormalized[1]	Normalized
Two-Dimensional Routines		
Input	Two-dimensional array	Multiple vector two-dimensional routines are not supported
Output	Two-dimensional transform or inverse transform	
Results	Unnormalized	
Three-Dimensional Routines		
Input	Three-dimensional array	Multiple vector three-dimensional routines are not supported
Output	Three-dimensional transform or inverse transform	
Results	Unnormalized	

1. Results of inverse transform must be divided by a normalization factor proportional to *N*.

Extensions to FFTPACK and VFFTPACK

Sun Performance Library provides the following extensions to the standard Netlib FFTPACK and VFFTPACK packages.

- Double precision and double complex transforms. Because routines that process double precision and double complex data are not available in the standard package from Netlib, calls to these routines might not be portable.
- Two-dimensional and three-dimensional FFTs. Netlib FFTPACK and VFFTPACK routines support one-dimensional FFTs.
- Convolution and correlation routines.
- Fortran 95 and C interfaces to FFTPACK and VFFTPACK.
- Optimizations for specific SPARC instruction set architectures.
- Support for a 64-bit enabled Solaris™ operating environment.
- Support for parallel processing compiler options.
- Support for multiple processor hardware options.

The Discrete Fourier Transform (DFT)

The FFT and VFFT routines provide an efficient means of computing the complex or real discrete Fourier transform and the discrete Fourier sine transform or discrete Fourier cosine transform of a real symmetric sequence.

The following definition of the DFT is used when calculating the complex discrete Fourier transform of a periodic sequence, where $i = \sqrt{-1}$.

$$X_k = \sum_{n=1}^{N} x_n e^{-i2\pi(n-1)(k-1)/N}, \qquad k = 1, \ldots, N$$

When calculating the inverse complex discrete Fourier transform, the following definition is used.

$$x_n = \sum_{k=1}^{N} X_k e^{i2\pi(n-1)(k-1)/N}, \qquad n = 1, \ldots, N$$

The results on the inverse transform are unnormalized and can be normalized by dividing each value by N.

When computing the DFT of a real sequence, the resulting array of Fourier coefficients is conjugate symmetric, where $X_k^* = X_{N-k-2}$ for k > 1, when using a one-based notation, or $X_k^* = X_{N-k}$ for k > 0, when using a zero-based notation. The asterisk * denotes complex conjugation, where $(a+ib)^* = a-ib$. The number of calculations required to compute the DFT is reduced by taking advantage of this symmetry.

When computing the transform of a real sequence, the complex discrete Fourier transform can be rewritten in the real trigonometric form shown in TABLE 5-2. In TABLE 5-2, $X_{(2k-2)}$ equals the real part of X_k, $X_{(2k-1)}$ equals the imaginary part of X_k, and X_N equals the real part of $X_{(N/2)+1}$.

TABLE 5-2 Formulas for Real FFT Routines

	Transform
Odd *N*	For $k = 2, \ldots, (N+1)/2$, $$X_1 = \sum_{n=1}^{N} x_n$$ $$X_{(2k-2)} = \sum_{n=1}^{N} x_n \cos\left(\frac{(k-1)(n-1)2\pi}{N}\right)$$ $$X_{(2k-1)} = \sum_{n=1}^{N} -x_n \sin\left(\frac{(k-1)(n-1)2\pi}{N}\right)$$
Even *N*	For $k = 2, \ldots, N/2$, $$X_1 = \sum_{n=1}^{N} x_n$$ $$X_{(2k-2)} = \sum_{n=1}^{N} x_n \cos\left(\frac{(k-1)(n-1)2\pi}{N}\right)$$ $$X_{(2k-1)} = \sum_{n=1}^{N} -x_n \sin\left(\frac{(k-1)(n-1)2\pi}{N}\right)$$ $$X_N = \sum_{n=1}^{N} (-1)^{(n-1)} x_n$$

TABLE 5-2 Formulas for Real FFT Routines *(Continued)*

	Inverse Transform
Odd *N*	For $n = 1, \ldots, N$, $x_n = X_1 + \sum_{k=2}^{(N+1)/2} \left(2X_{(2k-2)}\cos\left(\frac{(k-1)(n-1)2\pi}{N}\right) - 2X_{(2k-1)}\sin\left(\frac{(k-1)(n-1)2\pi}{N}\right)\right)$
Even *N*	For $n = 1, \ldots, N$, $x_n = X_1 + \sum_{k=2}^{N/2} \left(2X_{(2k-2)}\cos\left(\frac{(k-1)(n-1)2\pi}{N}\right) - 2X_{(2k-1)}\sin\left(\frac{(k-1)(n-1)2\pi}{N}\right)\right) + (-1)^{n-1}X_N$

The FFT routines can be used to compute the discrete Fourier cosine transform, discrete Fourier sine transform, and inverse transforms of the functions listed in TABLE 5-3.

TABLE 5-3 Symmetries Supported by FFT and VFFT Routines

Symmetry	Definition	Trigonometric Expansion
Cosine Even-Wave	An even function *f*(*t*) that satisfies the condition *f*(-*t*) = *f*(*t*).	Trigonometric series containing only cosine terms.
Cosine Quarter-Wave	A even function with half-wave symmetry $f(t) = -f(t + T/2)$, where T is the period of the function.	Trigonometric series containing only cosine terms with odd wave numbers.
Sine Odd-Wave	An odd function *f*(*t*) that satisfies the condition *f*(-*t*) =-*f*(*t*).	Trigonometric series containing only sine terms.
Sine Quarter-Wave	A odd function with half-wave symmetry $f(t) = -f(t + T/2)$.	Trigonometric series containing only sine terms with odd wave numbers.

The formulas for the symmetries listed in TABLE 5-3 are shown in TABLE 5-4.

TABLE 5-4 Formulas for Symmetries Supported by FFT and VFFT Routines

	Cosine Even-Wave[1]	
Transform/ Inverse Transform	$X_k = x_1 + 2\sum_{n=1}^{N-1} x_n \cos\left(\frac{(k-1)(n-1)\pi}{N-1}\right) + (-1)^{(k-1)} x_N$,	$k = 1, \ldots, N$
	Cosine Quarter-Wave	
Transform	$X_k = x_1 + 2\sum_{n=2}^{N} x_n \cos\left(\frac{(2k-1)(n-1)\pi}{2N}\right)$,	$k = 1, \ldots, N$
Inverse Transform	$x_n = 4\sum_{k=1}^{N} X_k \cos\left(\frac{(2k-1)(n-1)\pi}{2N}\right)$,	$n = 1, \ldots, N$
	Sine Odd-Wave[1]	
Transform/ Inverse Transform	$X_k = 2\sum_{n=1}^{N} x_n \sin\left(\frac{kn\pi}{(N+1)}\right)$,	$k = 1, \ldots, N$
	Sine Quarter-Wave	
Transform	$X_k = 2\sum_{n=1}^{N-1} x_n \sin\left(\frac{(2k-1)n\pi}{2N}\right) + (-1)^{(k-1)} x_N$,	$k = 1, \ldots, N$
Inverse Transform	$x_n = 4\sum_{k=1}^{N} X_k \sin\left(\frac{(2k-1)n\pi}{2N}\right)$,	$n = 1, \ldots, N$

1. The cosine even-wave and sine odd-wave routines perform either the transform or inverse transform, depending upon whether the input array contains the Fourier coefficients or the periodic sequence.

For additional information on the formulas used to calculate the discrete transforms of symmetric sequences, see the documentation provided with FFTPACK, available on Netlib at `http://www.netlib.org/fftpack/doc`.

Naming Conventions

The name of each FFT or VFFT routine is made up of a base name that denotes the operation performed and a prefix that denotes the operand data type. For example, the routine CFFTF performs a fast Fourier transform of a complex sequence.

Prefixes used with FFT and VFFT base names are shown in TABLE 5-5.

TABLE 5-5 Prefix and Operand Data Types

	Prefix	Operand Data Type
FFT Routines	No prefix	REAL, REAL*4, REAL(4)
	R	REAL, REAL*4, REAL(4)
	D	DOUBLE, REAL*8, REAL(8)
	C	COMPLEX, COMPLEX*8, COMPLEX(4)
	Z	DOUBLE COMPLEX, COMPLEX*16, COMPLEX(8)
VFFT Routines	VR	REAL, REAL*4, REAL(4)
	VD	DOUBLE, REAL*8, REAL(8)
	VC	COMPLEX, COMPLEX*8, COMPLEX(4)
	VZ	DOUBLE COMPLEX, COMPLEX*16, COMPLEX(8)

FFT and VFFT base names are shown in TABLE 5-6 on page 73. The last character of the base name is one of the following:

- I: Initialize the Fourier transform or inverse Fourier transform routine
- F: Compute the forward transform (the Fourier transform)
- B: Compute the backward transform (the inverse Fourier transform or synthesis)

TABLE 5-6 FFT and VFFT Base Names

Base Name	Operation
COSQB	Inverse cosine quarter-wave transform (synthesis)
COSQF	Cosine quarter-wave transform
COSQI	Initialize cosine quarter-wave transform or inverse transform
COST	Cosine even-wave transform
COSTI	Initialize cosine even-wave transform
EZFFTB	Inverse EZ transform (synthesis)
EZFFTF	EZ transform
EZFFTI	Initialize EZ transform
FFTB	Inverse transform (synthesis)
FFTF	Forward transform
FFTI	Initialize before computing a transform or inverse transform
SINQB	Inverse sine quarter-wave transform (synthesis)
SINQF	Sine quarter-wave transform
SINQI	Initialize sine quarter-wave transform or inverse transform
SINT	Sine odd-wave transform
SINTI	Initialize sine odd-wave transform

In this manual, the following conventions are used when referring to routines that exist for multiple data types:

- The prefix *x* is added to the base name when the information applies to `REAL`, `DOUBLE`, `COMPLEX`, and `DOUBLE COMPLEX` versions of that routine.
- Specific prefixes are listed in square brackets [ ] before the base name when information does not apply to all versions of the routine.

The following example shows samples of these naming conventions.

Convention	Routines
x`FFTF`	`RFFTF`, `DFFTF`, `CFFTF`, and `ZFFTF`
`[R,D]FFTI`	`RFFTI` or `DFFTI`
`[C,Z]FFTF`	`CFFTF` or `ZFFTF`
`V[R,D,C,Z]FFTF`	`VRFFTF`, `VDFFTF`, `VCFFTF`, or `VZFFTF`

Sun Performance Library FFT Routines

Sun Performance Library contains the routines shown in TABLE 5-8. The data type of the arguments follows the conventions shown in TABLE 5-7.

TABLE 5-7 Argument Data Types

Argument	Data Type
AZERO, A, B, R (EZFFT routines)	Real
FULL, PLACE, ROWCOL	Character
N, M, K, LDA, LD2A, LDB, LWORK, MDIMX	Integer
A, B, X, XT	Same as data type of routine called
WSAVE, WORK	See TABLE 5-10 on page 79

TABLE 5-8 FFT Routines

Routine	Arguments	Function
COSQB, DCOSQB	N,X,WSAVE	Inverse cosine quarter-wave transform
VCOSQB, VDCOSQB	M,N,X,XT,MDIMX,WSAVE	Inverse cosine quarter-wave transform (Vector)
COSQF, DCOSQF	N,X,WSAVE	Cosine quarter-wave transform
VCOSQF, VDCOSQF	M,N,X,XT,MDIMX,WSAVE	Cosine quarter-wave transform (Vector)
COSQI, DCOSQI	N,WSAVE	Initialize cosine quarter-wave transform and inverse transform
VCOSQI, VDCOSQI	N,WSAVE	Initialize cosine quarter-wave transform and inverse transform (Vector)
COST, DCOST	N,X,WSAVE	Cosine even-wave transform
VCOST, VDCOST	M,N,X,XT,MDIMX,WSAVE	Cosine even-wave transform (Vector)
COSTI, DCOSTI	N,WSAVE	Initialize cosine even-wave transform
VCOSTI, VDCOSTI	N,WSAVE	Initialize cosine even-wave transform (Vector)

TABLE 5-8 FFT Routines *(Continued)*

Routine	Arguments	Function
EZFFTB	N,R,AZERO,A,B,WSAVE	EZ inverse Fourier transform
EZFFTF	N,R,AZERO,A,B,WSAVE	EZ Fourier transform
EZFFTI	N,WSAVE	Initialize EZ Fourier transform and inverse transform
RFFTB, DFFTB, CFFTB, ZFFTB	N,X,WSAVE	Inverse Fourier transform
VRFFTB, VDFFTB	M,N,X,XT,MDIMX,WSAVE	Inverse Fourier transform (Vector)
VCFFTB, VZFFTB	M,N,X,XT,MDIMX, ROWCOL,WSAVE	
RFFTF, DFFTF, CFFTF, ZFFTF	N,X,WSAVE	Fourier transform
VRFFTF, VDFFTF	M,N,X,XT,MDIMX,WSAVE	Fourier transform (Vector)
VCFFTF, VZFFTF	M,N,X,XT,MDIMX, ROWCOL,WSAVE	
RFFTI, DFFTI, CFFTI, ZFFTI	N,WSAVE	Initialize Fourier transform and inverse transform
VRFFTI,VDFFTI, VCFFTI, VZFFTI	N,WSAVE	Initialize Fourier transform and inverse transform (Vector)
SINQB, DSINQB	N,X,WSAVE	Inverse sine quarter-wave transform
VSINQB, VDSINQB	M,N,X,XT,MDIMX,WSAVE	Inverse sine quarter-wave transform (Vector)
SINQF, DSINQF	N,X,WSAVE	Sine quarter-wave transform
VSINQF, VDSINQF	M,N,X,XT,MDIMX,WSAVE	Sine quarter-wave transform (Vector)
SINQI, DSINQI	N,WSAVE	Initialize sine quarter-wave transform and inverse transform
VSINQI, VDSINQI	N,WSAVE	Initialize sine quarter-wave transform and inverse transform (Vector)
SINT, DSINT	N,X,WSAVE	Sine odd-wave transform
VSINT, VDSINT	M,N,X,XT,MDIMX,WSAVE	Sine odd-wave transform (Vector)
SINTI, DSINT	N,WSAVE	Initialize sine odd-wave transform
VSINTI, VDSINTI	N,WSAVE	Initialize sine odd-wave transform (Vector)

TABLE 5-8 FFT Routines *(Continued)*

Routine	Arguments	Function
`RFFT2B`, `DFFT2B`	`PLACE,M,N,A,LDA, B,LDB,WORK,LWORK`	Inverse two-dimensional Fourier transform
`CFFT2B`, `ZFFT2B`	`M,N,A,LDA,WORK,LWORK`	
`RFFT2F`, `DFFT2F`	`PLACE,FULL,M,N,A,LDA, B,LDB,WORK,LWORK`	Two-dimensional Fourier transform
`CFFT2F`, `ZFFT2F`	`M,N,A,LDA,WORK,LWORK`	
`RFFT2I`,`DFFT2I`, `CFFT2I`, `ZFFT2I`	`M,N,WORK`	Initialize two-dimensional Fourier transform and inverse transform
`RFFT3B`, `DFFT3B`	`PLACE,M,N,K,A,LDA, B,LDB,WORK,LWORK`	Inverse three-dimensional Fourier transform
`CFFT3B`, `ZFFT3B`	`M,N,K,A,LDA,LD2A, WORK,LWORK`	
`RFFT3F`, `DFFT3F`	`PLACE,FULL,M,N,K, A,LDA,B,LDB,WORK,LWORK`	Three-dimensional Fourier transform
`CFFT3F`, `ZFFT3F`	`M,N,K,A,LDA,LD2A, WORK,LWORK`	
`RFFT3I`,`DFFT3I`, `CFFT3I`, `ZFFT3I`	`M,N,K,WORK`	Initialize three-dimensional Fourier transform and inverse transform

Sequence Length *N*

The efficiency of the FFT computation depends upon the length *N* of the input data set. The FFT computation is most efficient if *N* can be decomposed into one or more factors for which the Sun Performance Library contains highly optimized transform routines.

It is desirable if *N* can be factored into 2, 3, 4, or 5 for real-to-complex and complex-to-real transforms and into 2, 3, 4, 5, 7, 11, or 13 for complex-to- complex transforms. These transforms are of order *N* log_2 *N*. However, if *N* is a large prime or cannot be factored into the above values, the computation is of order N^2.

Computing the Fourier transform, fast cosine transform, fast sine transform, or multi-dimensional FFT is most efficient when the sequence length can be factored into powers of the supported prime factors, as summarized in TABLE 5-9.

TABLE 5-9 Values That Must Have 2, 3, 4, 5, 7, 11, or 13 as Factors for Best Performance

Routine	Values
`COST`, `DCOST`, `VCOST`, `VDCOST`	`N - 1`
`SINT`, `DSINT`, `VSINT`, `VDSINT`	`N + 1`
All other one-dimensional FFT and VFFT routines	`N`
Two-dimensional FFT routines	`M` and `N`
Three-dimensional FFT routines	`M`, `N`, and `K`

The function *x*`FFTOPT` can be used to determine the optimal sequence length, as shown in CODE EXAMPLE 5-1.

CODE EXAMPLE 5-1 `RFFTOPT` Example

```
my_system% cat fft_ex01.f
      PROGRAM TEST
      INTEGER         N, N1, N2, N3, RFFTOPT
C
      N = 1024
      N1 = 1019
      N2 = 71
      N3 = 49
C
      PRINT *, 'N Original  N Suggested'
      PRINT '(I5, I12)', (N, RFFTOPT(N))
      PRINT '(I5, I12)', (N1, RFFTOPT(N1))
      PRINT '(I5, I12)', (N2, RFFTOPT(N2))
      PRINT '(I5, I12)', (N3, RFFTOPT(N3))
      END

my_system% f95 -dalign fft_ex01.f -xlic_lib=sunperf
my_system% a.out
 N Original  N Suggested
 1024        1024
 1019        1024
   71          72
   49          49
```

The size of the sequence affects performance. When *N* is small, such as 8 or 16, the overhead of calling the routine is large compared to the actual computational work performed by the routine. Also, when the size of *N* is too large for the data to fit in the cache, performance again degrades.

Work Array WSAVE for FFT and VFFT Routines

Each FFT or VFFT routine uses a work array that stores the tabulation of trigonometric functions computed while generating the Fourier transform or inverse transform. WSAVE also stores scratch (temporary) values generated during the transform or inverse transform.

Note – When using the VFFT routines, an extra work array, XT, is used to store temporary values generated from performing Fourier transforms or inverse transforms on multiple sequences.

Before performing the first transform or inverse transform:

1. **Specify the minimum dimension and data type of the work array WSAVE.**

   The minimum dimension and data type depends upon the operand data type and FFT or VFFT routine, as shown in TABLE 5-10 on page 79.

2. **Initialize the work array by calling the corresponding FFT or VFFT routine whose base name ends with the character I.**

   For example, when using RFFTF or RFFTB, initialize the work array by calling RFFTI.

```
INTEGER N
REAL WSAVE (2 * N + 15)
CALL RFFTI (N, WSAVE)
```

When using CFFTF or CFFTB, initialize the work array by calling CFFTI.

```
INTEGER N
REAL WSAVE (4 * N + 15)
CALL CFFTI (N, WSAVE)
```

The same work array can be used for both the transform or inverse transform as long as N remains unchanged. Different WSAVE arrays are required for different values of N. As long as N and WSAVE remain unchanged, subsequent transforms can be obtained faster than the first transform, because the initialization does not have to be repeated between calls to the transform or inverse transform routines.

TABLE 5-10 Minimum Dimensions and Data Types for WSAVE Work Array

Routine	Minimum Work Array Size (WSAVE)	Type
One-Dimensional Routines		
COSQI, DCOSQI	3N + 15	REAL, REAL*8
COSQB, DCOSQB	3N + 15	REAL, REAL*8
COSQF, DCOSQF	3N + 15	REAL, REAL*8
COST, DCOST	3N + 15	REAL, REAL*8
COSTI, DCOSTI	3N + 15	REAL, REAL*8
EZFFTB	3N + 15	REAL, REAL*8
EZFFTF	3N + 15	REAL, REAL*8
EZFFTI	3N + 15	REAL, REAL*8
RFFTB, DFFTB	2N + 15	REAL, REAL*8
RFFTF, DFFTF	2N + 15	REAL, REAL*8
RFFTI, DFFTI	2N + 15	REAL, REAL*8
CFFTB, ZFFTB	4N + 15	REAL, REAL*8
CFFTF, ZFFTF	4N + 15	REAL, REAL*8
CFFTI, ZFFTI	4N + 15	REAL, REAL*8
SINQB, DSINQB	3N + 15	REAL, REAL*8
SINQF, DSINQF	3N + 15	REAL, REAL*8
SINQI, DSINQI	3N + 15	REAL, REAL*8
SINT, DSINT	2N + N/2 + 15	REAL, REAL*8
SINTI, DSINTI	2N + N/2 + 15	REAL, REAL*8
VFFT Routines		
VRFFTB, VDFFTB	N + 15	REAL, REAL*8
VRFFTF, VDFFTF	N + 15	REAL, REAL*8
VRFFTI, VDFFTI	N + 15	REAL, REAL*8
VCFFTB, VZFFTB	If transforming rows: 2 * M + 15 If transforming columns: 2 * N + 15	REAL, REAL*8

TABLE 5-10 Minimum Dimensions and Data Types for `WSAVE` Work Array *(Continued)*

Routine	Minimum Work Array Size (`WSAVE`)	Type
`VCFFTF, VZFFTF`	If transforming rows: `2 * M + 15` If transforming columns: `2 * N + 15`	`REAL, REAL*8`
`VCFFTI, VZFFTI`	`N + 15`	`REAL, REAL*8`
`VCOSQB, VDCOSQB`	`2 * N + 15`	`REAL, REAL*8`
`VCOSQF, VDCOSQF`	`2 * N + 15`	`REAL, REAL*8`
`VCOSQI, VDCOSQI`	`2 * N + 15`	`REAL, REAL*8`
`VCOST, VDCOST`	`2 * N + 15`	`REAL, REAL*8`
`VCOSTI, VDCOSTI`	`2 * N + 15`	`REAL, REAL*8`
`VSINQB, VDSINQB`	`2 * N + 15`	`REAL, REAL*8`
`VSINQF, VDSINQF`	`2 * N + 15`	`REAL, REAL*8`
`VSINQI, VDSINQI`	`2 * N + 15`	`REAL, REAL*8`
`VSINT, VDSINT`	`N + N/2 + 15`	`REAL, REAL*8`
`VSINTI, VDSINTI`	`N + N/2 + 15`	`REAL, REAL*8`
Two-Dimensional Routines		
`RFFT2B, DFFT2B`	`(M + 2N + MAX(M, 2N) + 30)`	`REAL, REAL*8`
`RFFT2F, DFFT2F`	`(M + 2N + MAX(M, 2N) + 30)`	`REAL, REAL*8`
`RFFT2I, DFFT2I`	`(M + 2N + MAX(M, 2N) + 30)`	`REAL, REAL*8`
`CFFT2B, ZFFT2B`	`(4 * (M + N) + 30)`	`REAL, REAL*8`
`CFFT2F, ZFFT2F`	`(4 * (M + N) + 30)`	`REAL, REAL*8`
`CFFT2I, ZFFT2I`	`(4 * (M + N) + 30)`	`REAL, REAL*8`
Three-Dimensional Routines		
`RFFT3B, DFFT3B`	`(M + 2 * (N + K) + 4K + 45)`	`REAL, REAL*8`
`RFFT3F, DFFT3F`	`(M + 2 * (N + K) + 4K + 45)`	`REAL, REAL*8`
`RFFT3I, DFFT3I`	`(M + 2 * (N + K) + 30)`	`REAL, REAL*8`
`CFFT3B, ZFFT3B`	`(4 * (M + N + K) + 45)`	`REAL, REAL*8`
`CFFT3F, ZFFT3F`	`(4 * (M + N + K) + 45)`	`REAL, REAL*8`
`CFFT3I, ZFFT3I`	`(4 * (M + N + K) + 45)`	`REAL, REAL*8`

One-Dimensional FFT and Inverse Transform Routines

The routines in this section use the fast Fourier transform to compute the discrete Fourier transform and the inverse Fourier transforms. Routines are also available that compute the fast cosine transform, fast sine transform, and the inverses of these transforms.

Arguments for One-Dimensional FFT and VFFT Routines

FFT and VFFT routines use the arguments shown in TABLE 5-11. Some routines use additional arguments that are described in the sections for those routines.

TABLE 5-11 Arguments for FFT and VFFT Routines

Arguments	Description
FFT Routines	
`N`	Length of the sequence to be transformed, where `N` ≥ 0.
`X`	On entry, an array of length `N` containing the sequence to be transformed.
`WSAVE`	On entry, a work array with a minimum dimension that depends upon the type of routine used and the data type of the operands. See TABLE 5-10 for a complete list of minimum work array dimensions.

TABLE 5-11 Arguments for FFT and VFFT Routines *(Continued)*

Arguments	Description
VFFT Routines	
`N`	Length of the sequence to be transformed, where `N` ≥ 0.
`M`	Number of sequences to be transformed, where `M` ≥ 0.
`X`	A two-dimensional array `X(M,N)` whose rows contain the sequences to be transformed.
`XT`	A two-dimensional work array with dimensions of `(MDIMX * N)`.
`MDIMX`	Leading dimension of the arrays `X` and `XT` as specified in a dimension or type statement, where `MDIMX` ≥ `M`.
`WSAVE`	On entry, a work array with a minimum dimension that depends upon the type of routine used and the data type of the operands. See TABLE 5-10 for a complete list of minimum work array dimensions.

Data Storage for One-Dimensional FFT and VFFT Routines

The data storage format for the computed Fourier coefficients depends upon whether the sequence is complex or real.

Storage of Complex Sequences

The results of a complex one-dimensional FFT are stored in-place (in the original input array). Storage problems do not occur when performing the Fourier transform of a complex sequence, because the number of calculated Fourier coefficients equals the number of input values. The real and imaginary values of the Fourier coefficients are stored in the original complex array without additional storage manipulations.

Storage of Real Sequences

Computing the Fourier transform of a real sequence produces complex Fourier coefficients. The number of computed Fourier coefficients is twice the number of values in the original sequence, because of the real and imaginary parts of the complex Fourier coefficients. The complex vector must be packed before it can be stored in the original real array. This packing is done by not storing the imaginary parts of the one or two Fourier coefficients that are always 0, and by not storing the complex conjugates of the Fourier coefficients.

Given a real sequence x_n, $n = 0 : N - 1$, of N data points, the transformed output X_k, $k = 0 : N - 1$, is packed and stored in the original array that holds the input data, as follows.

- If N is even:
  - The real part of X_0 is stored.
  - The imaginary part of X_0 is equal to 0; this part is not stored.
  - The real and imaginary parts of X_1, up to and including the real part of $X_{(N/2)}$, are stored sequentially.
  - The imaginary part of $X_{(N/2)}$ is equal to 0; this part is not stored.
  - $X_{(N-k)}$ is the complex conjugate of X_k, for $k = 1 : N/2 - 1$ and is not stored.
- If N is odd:
  - The real part of X_0 is stored.
  - The imaginary part of X_0 is equal to 0 and is not stored.
  - The real and imaginary parts of X_1, up to and including the imaginary part of $X_{((N-1)/2)}$, are stored sequentially.
  - $X_{(N-k)}$ is the complex conjugate of X_k, for $k = 1 : ((N-1)/2) - 1$ and is not stored.

For example, if N = 6, the input array `X` contains the following six real data points:

`X(1)` = x_0

`X(2)` = x_1

`X(3)` = x_2

`X(4)` = x_3

`X(5)` = x_4

`X(6)` = x_5

Performing the Fourier transform computes the complex Fourier coefficients X_0, X_1, X_2, X_3, X_4, and X_5, each of which has a real (Re) part and an imaginary (Im) part. Following the transform, the complex Fourier coefficients are stored in the original real array `X`, as follows:

`X(1)` = $\mathrm{Re}(X_0)$

`X(2)` = $\mathrm{Re}(X_1)$

`X(3)` = $\mathrm{Im}(X_1)$

`X(4)` = $\mathrm{Re}(X_2)$

`X(5)` = $\mathrm{Im}(X_2)$

`X(6)` = $\mathrm{Re}(X_3)$

For even-length vectors, the resulting vector is conjugate-symmetric excluding the first element. The Fourier transform of the vector [1 2 3 4] is:

10+0*i* -2+2*i* -2+0*i* -2-2*i*

This is stored in a real vector as:

10 -2 2 -2

For odd-length vectors, the resulting vector is also conjugate-symmetric excluding the first element. For example, the Fourier transform of the vector [1 2 3 4 5] is:

15.0+0*i* -2.5+3.44*i* -2.5+.81*i* -2.5-.81*i* -2.5-3.44*i*

This is stored in a real vector as:

15 -2.5 3.44 -2.5 0.81

Note – When the transform of complex data is computed, the output is not packed. The transformed sequence contains the same number of real and complex values as the input sequence.

CODE EXAMPLE 5-2 computes the FFT and inverse of a real or complex sequence for even and odd values of *N*. The transform of the complex sequence shows all the Fourier coefficients in an unpacked, complex array. The transform of the real sequence shows the Fourier coefficients stored in a packed, real array. Differences between the real arrays for even and odd values of *N* can also be compared.

CODE EXAMPLE 5-2 Real and Complex FFT Example

```
my_system% cat fft_ex02.f
      INTEGER I, N_EVEN, N_ODD
      REAL XR(9), WORK(1000)
      COMPLEX XC(9)
      N_EVEN = 8
      N_ODD = 9
      XR(1:N_EVEN) = (/.60,.25,.74,.26,.14,.93,.28,.04/)
      XC(1:N_EVEN) = (/.60,.25,.74,.26,.14,.93,.28,.04/)
C
      CALL RFFTI(N_EVEN, WORK)
      CALL RFFTF(N_EVEN, XR, WORK)
      CALL CFFTI(N_EVEN, WORK)
      CALL CFFTF(N_EVEN, XC, WORK)
```

CODE EXAMPLE 5-2 Real and Complex FFT Example *(Continued)*

```
      PRINT 1000
      PRINT '(F8.3)',XR(1:N_EVEN)
      PRINT 1010
      PRINT '(2F8.3,''I'')', (XC(1:N_EVEN))
      XR(1:N_ODD) = (/.60,.25,.74,.26,.14,.93,.28,.04,.02/)
      XC(1:N_ODD) = (/.60,.25,.74,.26,.14,.93,.28,.04,.02/)
C
      CALL RFFTI(N_ODD, WORK)
      CALL RFFTF(N_ODD, XR, WORK)
      CALL CFFTI(N_ODD, WORK)
      CALL CFFTF(N_ODD, XC, WORK)
      PRINT 1020
      PRINT '(F8.3)',XR(1:N_ODD)
      PRINT 1030
      PRINT '(2F8.3,''I'')', (XC(1:N_ODD))
C
 1000 FORMAT (1X, "Transform of Real Sequence With Even N")
 1010 FORMAT (1X, "Transform of Complex Sequence With Even N")
 1020 FORMAT (1X, "Transform of Real Sequence With Odd N")
 1030 FORMAT (1X, "Transform of Complex Sequence With Odd N")
      END
my_system% f95 -dalign fft_ex02.f -xlic_lib=sunperf
my_system% a.out
 Transform of real sequence with even N
   3.240
  -0.176
  -0.135
  -0.280
  -0.880
   1.096
   0.785
   0.280
 Transform of complex sequence with even N
    3.240   0.000i
   -0.176  -0.135i
   -0.280  -0.880i
    1.096   0.785i
    0.280   0.000i
    1.096  -0.785i
   -0.280   0.880i
   -0.176   0.135i
```

CODE EXAMPLE 5-2 Real and Complex FFT Example *(Continued)*

```
Transform of real sequence with odd N
  3.260
 -0.333
 -0.550
  0.464
 -0.991
  0.080
  1.091
  0.860
 -0.389
Transform of complex sequence with odd N
   3.260   0.000i
  -0.333  -0.550i
   0.464  -0.991i
   0.080   1.091i
   0.860  -0.389i
   0.860   0.389i
   0.080  -1.091i
   0.464   0.991i
  -0.333   0.550i
```

CODE EXAMPLE 5-3 on page 87 shows a C example that uses `dfftf` to compute the Fourier coefficients of a real sequence.

CODE EXAMPLE 5-3 C Example Showing How to Extract the Complex Result From the Packed Output of `dfftf`

```
my_system% cat fft_ex03.c
#include <sunperf.h>
#include <math.h>

#define N 16

/*
 dfftf accepts as input a real vector of length N and
 computes its discrete Fourier transform. Since the input
 is real, the result of the transform will be conjugate symmetric.
 The output of dfftf is a real vector of length N, which is a
 packed representation of the complex FFT result. Only the first
 half of the complex result is stored since the remaining values
 can be obtained via the conjugate symmetry property. In
 particular, if A[N] is the complex result of the FFT, the output
 of dfftf is related to ‘a’ as follows:
 The real part of A[0] is stored in a[0].
 A[1] is stored as two consecutive real numbers in a[1] and a[2].
 A[2] is stored in a[3] and a[4].
 If N is even, the real part of A[N/2-1] is stored in a[N-1]. If
 N is odd, the real and imaginary parts of A[(N-1)/2] are stored
 in a[N-2] and a[N-1] respectively.
 The following example shows how to extract the complex result
 from the packed output of dfftf for the case in which N even.
*/

void
main()
{
  int   i,j;
  double a[N];
  doublecomplex b[N];

  double wa[2*N+15];

  for (i=0;i<N;i++) {
    a[i]=sin((double)i);
  }
  dffti(N,wa);
  dfftf(N,a,wa);
```

CODE EXAMPLE 5-3 C Example Showing How to Extract the Complex Result From the Packed Output of `dfftf` *(Continued)*

```
  /* extract the first N/2 complex values
     from the packed representation */

  b[0].r = a[0];
  b[0].i = 0.0;

  j=1;
  for (i=1;i<N/2;i++) {
    b[i].r = a[j];
    b[i].i = a[j+1];
    j += 2;
  }
  b[N/2].r = a[N-1];
  b[N/2].i = 0.0;

  /* extract the remaining N/2 values using the conjugate
     symmetry */

  for (i=N/2+1;i<N;i++) {
    b[i].r =  b[N-i].r;
    b[i].i = -b[N-i].i;
  }
}
```

FFT: Fast Fourier Transform Routines

The following routines use the fast Fourier transform to compute the discrete Fourier transform or inverse transform of a periodic sequence.

Routine	Function
`[R,D,C,Z]FFTI`	Initialize work array `WSAVE` for `[R,D,C,Z]FFTF` or `[R,D,C,Z]FFTB`
`[R,D,C,Z]FFTF`	Compute Fourier coefficients of periodic sequence
`[R,D,C,Z]FFTB`	Compute periodic sequence from Fourier coefficients
`V[R,D,C,Z]FFTI`	Initialize work array for `V[R,D,C,Z]FFTF` or `V[R,D,C,Z]FFTB`
`V[R,D,C,Z]FFTF`	Compute Fourier coefficients of multiple periodic sequences
`V[R,D,C,Z]FFTB`	Compute multiple periodic sequences from Fourier coefficients

The *x*FFT and V*x*FFT routines, where x denotes `R`, `D`, `C`, or `Z`, use the arguments defined in "Arguments for One-Dimensional FFT and VFFT Routines" on page 81.

In addition to the VFFT arguments defined in "Arguments for One-Dimensional FFT and VFFT Routines" on page 81, the `VCFFTF`, `VZFFTF`, `VCFFTB`, and `VZFFTB` routines use one additional argument called `ROWCOL.` `ROWCOL` specifies whether to transform the rows or columns of `X(M,N)`. Set `ROWCOL` equal to `'R'` or `'r'` perform the transform or inverse transform on the rows of `X(M,N)`. Set `ROWCOL` equal to `'C'` or `'c'` perform the transform or inverse transform on the columns of `X(M,N)`.

Normalization

The *x*FFT operations are unnormalized, so a call of *x*FFTF followed by a call of *x*FFTB will multiply the input sequence by N. The V*x*FFT operations are normalized, so a call of V*x*FFTF followed by a call of V*x*FFTB will return the original sequence.

Sample Programs: Fast Fourier Transform and Inverse Transform

CODE EXAMPLE 5-4 uses RFFTF to compute the FFT of a real sequence and RFFTB to compute the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is unnormalized and can be normalized by dividing each value by N.

CODE EXAMPLE 5-4 Fast Fourier Transform and Inverse Transform for Real Values

```
my_system% cat fft_ex04.f
      PROGRAM TEST
C
      INTEGER           N
      PARAMETER        (N = 9)
      INTEGER           I
      REAL              PI, R(N), WSAVE(2 * N + 15)
      EXTERNAL          RFFTB, RFFTF, RFFTI
      INTRINSIC         ACOS, SIN
C
C     Initialize array to a real sequence.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        R(I) = 3.0 + SIN ((I - 1.0) * 2.0 * PI / N)
  100 CONTINUE
C
      PRINT 1000
      PRINT 1010, (R(I), I = 1, N)
      CALL RFFTI (N, WSAVE)
      CALL RFFTF (N, R, WSAVE)
      PRINT 1020
      PRINT 1010, (R(I), I = 1, N)
      CALL RFFTB (N, R, WSAVE)
      PRINT 1030
      PRINT 1010, (R(I), I = 1, N)
C
 1000 FORMAT (1X, 'Original Sequence R(I): ')
 1010 FORMAT (1X, 100(F4.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Unnormalized Recovered Sequence (R(I)*N): ')
C
      END
```

CODE EXAMPLE 5-4 Fast Fourier Transform and Inverse Transform for Real Values

```
my_system% f95 -dalign fft_ex04.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence R(I):
  3.0  3.6  4.0  3.9  3.3  2.7  2.1  2.0  2.4
 Transformed Sequence:
 27.0  0.0 -4.5  0.0  0.0  0.0  0.0  0.0  0.0
 Unnormalized Recovered Sequence (R(I)*N):
 27.0 32.8 35.9 34.8 30.1 23.9 19.2 18.1 21.2
```

CODE EXAMPLE 5-5 uses `CFFTF` to compute the FFT of a complex sequence and `CFFTB` to compute the inverse transform. Because the number of calculated Fourier coefficients equals the number of input values, the real and imaginary values of the Fourier coefficients can be stored in the original array without additional storage manipulations. The inverse transform is unnormalized and can be normalized by dividing each value by *N*.

CODE EXAMPLE 5-5 Fast Fourier Transform and Inverse Transform for Complex Values

```
my_system% cat fft_ex05.f
      PROGRAM TEST
C
      INTEGER           N
      PARAMETER        (N = 4)
C
      INTEGER           I
      REAL              PI, WSAVE(4 * N + 15), X, Y
      COMPLEX           C(N)
C
      EXTERNAL          CFFTB, CFFTF, CFFTI
      INTRINSIC         ACOS, CMPLX, COS, SIN
C     Initialize the array C to a complex sequence.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X = SIN ((I - 1.0) * 2.0 * PI / N)
        Y = COS ((I - 1.0) * 2.0 * PI / N)
        C(I) = CMPLX (X, Y)
  100 CONTINUE
      PRINT 1000
      PRINT 1010, (C(I), I = 1, N)
```

CODE EXAMPLE 5-5 Fast Fourier Transform and Inverse Transform for Complex Values

```
      CALL CFFTI (N, WSAVE)
      CALL CFFTF (N, C, WSAVE)
      PRINT 1020
      PRINT 1010, (C(I), I = 1, N)
      CALL CFFTB (N, C, WSAVE)
      PRINT 1030
      PRINT 1010, (C(I), I = 1, N)
C
 1000 FORMAT (1X, 'Original Sequence C(I):')
 1010 FORMAT (1X, 100(F5.1, ' +',F4.1,'i  '))
 1020 FORMAT (1X, 'Transformed Sequence:')
 1030 FORMAT (1X, 'Unnormalized Recovered Sequence (C(I)*N):')
C
      END
my_system% f95 -dalign fft_ex05.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence C(I):
   0.0 + 1.0i    1.0 + 0.0i    0.0 +-1.0i   -1.0 + 0.0i
 Transformed Sequence:
   0.0 + 0.0i    0.0 + 0.0i    0.0 + 0.0i    0.0 + 4.0i
 Unnormalized Recovered Sequence (C(I)*N):
   0.0 + 4.0i    4.0 + 0.0i    0.0 +-4.0i   -4.0 + 0.0i
```

EZFFT: EZ Fourier Transform Routines

The following routines are used to perform a Fourier transform or inverse transform of a real periodic sequence. The EZ Fourier or inverse transform routines are simplified but slower versions of the Fast Fourier Transform routines.

Routine	Function
EZFFTI	Initialize work array WSAVE for EZFFTF or EZFFTB
EZFFTF	Compute Fourier coefficients of periodic sequence
EZFFTB	Compute periodic sequence from Fourier coefficients

The EZFFT routines use the arguments shown in TABLE 5-12.

TABLE 5-12 Arguments for EZFFT Routines

Argument	Definition
N	Sequence length
R	For EZFFTF, a real array containing the sequence to be transformed, unchanged on exit. For EZFFTB, a real array containing the Fourier coefficients of the inputs.
AZERO	The Fourier constant A_0
A	Real array containing the real parts of the complex Fourier coefficients. If N is even, then A is length N/2, otherwise A is length (N–1)/2.
B	Real array containing the imaginary parts of the complex Fourier coefficients. If N is even, then B is length N/2, otherwise B is length (N–1)/2.
WSAVE	Work array initialized by EZFFTI

Sample Program: EZ Fourier Transform and Inverse Transform

CODE EXAMPLE 5-6 uses EZFFTF to compute a Fourier transform of a real sequence and EZFFTB to compute the inverse transform. When using EZFFTF, the computed Fourier coefficients are stored in the arrays A and B. The input array R is not overwritten. Unlike the output of RFFTF and DFFTF, no packing is performed, and the complex conjugates are retained.

CODE EXAMPLE 5-6 EZ Fourier Transform and Inverse Transform

```
my_system% cat fft_ex06.f
      PROGRAM TEST
C
      INTEGER          N
      PARAMETER       (N = 9)
      INTEGER          I
      REAL             A(N), B(N), AZERO, PI, R(N)
      REAL             WSAVE(3 * N + 15)
      EXTERNAL         EZFFTB, EZFFTF, EZFFTI
      INTRINSIC        ACOS, COS, SIN
C
```

CODE EXAMPLE 5-6 EZ Fourier Transform and Inverse Transform *(Continued)*

```
C     Initialize array to a sequence of real numbers.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        R(I) = 3.0 + SIN ((I - 1.0) * 2.0 * PI / N)  +
     $         4.0 * COS ((I - 1.0) * 8.0 * PI / N)
  100 CONTINUE
C
      CALL EZFFTI (N, WSAVE)
      PRINT 1000
      PRINT 1010, (R(I), I = 1, N)
      CALL EZFFTF (N, R, AZERO, A, B, WSAVE)
      PRINT 1020, AZERO
      PRINT 1030
      PRINT 1010, (A(I), I = 1, N)
      PRINT 1040
      PRINT 1010, (B(I), I = 1, N)
      CALL EZFFTB (N, R, AZERO, A, B, WSAVE)
      PRINT 1050
      PRINT 1010, (R(I), I = 1, N)
C
 1000 FORMAT (1X, ’Original Sequence: ’)
 1010 FORMAT (100(F6.1, 1X))
 1020 FORMAT (1X, ’Azero = ’, F4.1)
 1030 FORMAT (1X, ’A =  ’)
 1040 FORMAT (1X, ’B = ’)
 1050 FORMAT (1X, ’Recovered Sequence: ’)
C
      END
my_system% f95 -dalign fft_ex06.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   7.0   -0.1    7.0    1.9    4.0    3.4    0.1    5.1   -1.4
 Azero =  3.0
 A =
   0.0    0.0    0.0    4.0    0.0    0.0    0.0    0.0    0.0
 B =
   1.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
 Recovered Sequence:
   7.0   -0.1    7.0    1.9    4.0    3.4    0.1    5.1   -1.4
```

COSQ: Cosine Quarter-Wave Routines

The following routines are used to perform a discrete Fourier cosine transform or inverse transform of a cosine series with only odd wave numbers.

Routine	Function
`[D]COSQI`	Initialize work array `WSAVE` for `[D]COSQF` or `[D]COSQB`
`[D]COSQF`	Compute Fourier coefficients of cosine series with odd wave numbers
`[D]COSQB`	Compute periodic sequence from Fourier coefficients
`V[D]COSQI`	Initialize work array for `V[D]COSQF` or `V[D]COSQB`
`V[D]COSQF`	Compute Fourier coefficients of multiple cosine series with odd wave numbers
`V[D]COSQB`	Compute multiple periodic sequences from Fourier coefficients

Because of the assumption of symmetry, the sequence used as input to the cosine quarter-wave routine only needs to contain the part of the sequence that is sufficient to determine the entire sequence.

Normalization

The *x*COSQ operations are unnormalized inverses of themselves, so a call to *x*COSQF followed by a call to *x*COSQB will multiply the input sequence by $4 \times N$. The V*x*COSQ operations are normalized, so a call of V*x*COSQF followed by a call of V*x*COSQB will return the original sequence.

Sample Programs: Cosine Quarter-Wave Transform and Inverse Transform

CODE EXAMPLE 5-7 on page 96 uses `COSQF` to compute the cosine quarter-wave transform of a real sequence and `COSQB` to compute the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is unnormalized and can be normalized by dividing each value by `4*N`.

CODE EXAMPLE 5-7 Cosine Quarter-Wave Transform and Inverse Transform

```
my_system% cat fft_ex07.f
      PROGRAM TEST
C
      INTEGER       N
      PARAMETER    (N = 6)
      INTEGER       I
      REAL          PI, WSAVE(3 * N + 15), X(N)
      EXTERNAL      COSQB, COSQF, COSQI
      INTRINSIC     ACOS, COS
C
C     Initialize array X to a real even quarter-wave sequence,
C     that is, it can be expanded in terms of a cosine series
C     with only odd wave numbers.
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(I) = COS((I - 1) * PI / (2.0 * N))
  100 CONTINUE
C
      CALL COSQI (N, WSAVE)
      PRINT 1000
      PRINT 1010, (X(I), I = 1, N)
      CALL COSQF (N, X, WSAVE)
      PRINT 1020
      PRINT 1010, (X(I), I = 1, N)
      CALL COSQB (N, X, WSAVE)
      PRINT 1030
      PRINT 1010, (X(I), I = 1, N)
C
 1000 FORMAT(1X, 'Original Sequence: ')
 1010 FORMAT(1X, 100(F7.3, 1X))
 1020 FORMAT(1X, 'Transformed Sequence: ')
 1030 FORMAT(1X, 'Recovered Sequence: ')
      END
my_system% f95 -dalign fft_ex07.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   1.000   0.966   0.866   0.707   0.500   0.259
 Transformed Sequence:
   6.000   0.000   0.000   0.000   0.000   0.000
 Recovered Sequence:
  24.000  23.182  20.785  16.971  12.000   6.212
```

CODE EXAMPLE 5-8 uses VCOSQF to compute the cosine quarter-wave transform of a single real sequence and VCOSQB to compute the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is normalized.

CODE EXAMPLE 5-8 Cosine Quarter-Wave Transform and Inverse Transform Using Vector Routines

```
my_system% cat fft_ex08.f
      PROGRAM TEST
C
      INTEGER          M, N
      PARAMETER       (M = 1)
      PARAMETER       (N = 6)
      INTEGER          I
      REAL             PI,  WSAVE(3 * N + 15), X(M, N), XT(M, N)
      EXTERNAL         VCOSQB, VCOSQF, VCOSQI
      INTRINSIC        ACOS, COS
C
C     Initialize the first row of the array to a real even
C     quarter-wave sequence, that is, it can be expanded in
C     terms of a cosine series with only odd wave numbers.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(M,I) = 40.0 * COS ((I - 1) * PI / (2.0 * N))
  100 CONTINUE
C
      PRINT 1000
      PRINT 1010, (X(M, I), I = 1, N)
      CALL VCOSQI (N, WSAVE)
      CALL VCOSQF (M, N, X, XT, M, WSAVE)
      PRINT 1020
      PRINT 1010, (X(M, I), I = 1, N)
      CALL VCOSQB (M, N, X, XT, M, WSAVE)
      PRINT 1030
      PRINT 1010, (X(M, I), I = 1, N)
C
 1000 FORMAT (1X, ’Original Sequence: ’)
 1010 FORMAT (1X, 100(F5.1, 1X))
 1020 FORMAT (1X, ’Transformed Sequence: ’)
 1030 FORMAT (1X, ’Recovered Sequence: ’)
      END
```

CODE EXAMPLE 5-8 Cosine Quarter-Wave Transform and Inverse Transform Using Vector Routines *(Continued)*

```
my_system% f95 -dalign fft_ex08.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
  40.0  38.6  34.6  28.3  20.0  10.4
 Transformed Sequence:
  49.0   0.0   0.0   0.0   0.0   0.0
 Recovered Sequence:
  40.0  38.6  34.6  28.3  20.0  10.4
```

CODE EXAMPLE 5-9 uses VCOSQF to compute the cosine quarter-wave transform of multiple real sequences and VCOSQB to compute the inverse transforms. The computed Fourier coefficients of each sequence are packed and stored in the rows of the original real array. The inverse transforms are normalized.

CODE EXAMPLE 5-9 Cosine Quarter-Wave Transform and Inverse Transform Using Vector Routines

```
my_system% cat fft_ex09.f
      PROGRAM TEST
      INTEGER           M, N
      PARAMETER        (M = 4)
      PARAMETER        (N = 6)
      INTEGER           I, J
      REAL              PI, WSAVE(N + 15), X(M, N), XT(M, N)
      EXTERNAL          VCOSQB, VCOSQF, VCOSQI
      INTRINSIC         ACOS, COS
C
C    Initialize the array to m real even quarter-wave sequences,
C     that is, they can be expanded in terms of a cosine series
C     with only odd wave numbers.
      PI = ACOS (-1.0)
      DO 110, J=1, M
        DO 100, I=1, N
          X(J,I) = 40.0 * J * COS ((I-1) * PI / 2.0 / N )
  100   CONTINUE
  110 CONTINUE
C
```

CODE EXAMPLE 5-9 Cosine Quarter-Wave Transform and Inverse Transform Using Vector Routines *(Continued)*

```
      CALL VCOSQI (N, WSAVE)
      PRINT 1000
      DO 120, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
  120 CONTINUE
      CALL VCOSQF (M, N, X, XT, M, WSAVE)
      PRINT 1020
      DO 130, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
 130  CONTINUE
      CALL VCOSQB (M, N, X, XT, M, WSAVE)
      PRINT 1030
      DO 140, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
 140  CONTINUE
C
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT(1X, '  Sequence', I2, ':  ', 100(F5.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
C
      END
my_system% f95 -dalign fft_ex09.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   Sequence 1:   40.0  38.6  34.6  28.3  20.0  10.4
   Sequence 2:   80.0  77.3  69.3  56.6  40.0  20.7
   Sequence 3:  120.0 115.9 103.9  84.9  60.0  31.1
   Sequence 4:  160.0 154.5 138.6 113.1  80.0  41.4
 Transformed Sequence:
   Sequence 1:   49.0   0.0   0.0   0.0   0.0   0.0
   Sequence 2:   98.0   0.0   0.0   0.0   0.0   0.0
   Sequence 3:  147.0   0.0   0.0   0.0   0.0   0.0
   Sequence 4:  196.0   0.0   0.0   0.0   0.0   0.0
 Recovered Sequence:
   Sequence 1:   40.0  38.6  34.6  28.3  20.0  10.4
   Sequence 2:   80.0  77.3  69.3  56.6  40.0  20.7
   Sequence 3:  120.0 115.9 103.9  84.9  60.0  31.1
   Sequence 4:  160.0 154.5 138.6 113.1  80.0  41.4
```

COST: Cosine Even-Wave Routines

The following routines are used to perform a discrete fourier cosine transform of an even sequence.

Routine	Function
`[D]COSTI`	Initialize work array `WSAVE` for `[D]COSTF` or `[D]COSTB`
`[D]COST`	Compute the Fourier coefficients or inverse transform of an even sequence
`V[D]COSTI`	Initialize work array for `V[D]COSTF` or `V[D]COSTB`
`V[D]COST`	Compute Fourier coefficients or inverse transform of multiple even sequences

The cosine even-wave routines are their own inverse. *x*COST computes the Fourier coefficients from a periodic sequence or the periodic sequence from the Fourier coefficients. *x*COSTF and *x*COSTB routines do not exist for cosine even-wave transforms.

Because of the assumption of symmetry, the sequence used as input to the cosine even-wave routine only needs to contain the part of the sequence that is sufficient to determine the entire sequence.

Normalization

The *x*COST transforms are unnormalized inverses of themselves, so a call of *x*COST followed by another call of *x*COST will multiply the input sequence by $2 \times (N-1)$. The V*x*COST transforms are normalized, so a call of V*x*COST followed by a call of V*x*COST will return the original sequence.

Sample Program: Cosine Even-Wave Transform and Inverse Transform

CODE EXAMPLE 5-10 on page 101 uses `COST` to compute the cosine even-wave transform of a real sequence and the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is unnormalized and can be normalized by dividing each value by `2*(N-1)`.

CODE EXAMPLE 5-10 Cosine Even-Wave Transform and Inverse Transform

```
my_system% cat fft_ex10.f
      PROGRAM TEST
C
      INTEGER          N
      PARAMETER       (N = 9)
      INTEGER          I
      REAL             PI, X(N), WSAVE(3 * N + 15)
      EXTERNAL         COST, COSTI
      INTRINSIC        ACOS, COS
C
C     Initialize the array X to an even sequence, that is, it
C     can be expanded in terms of a trigonometric series that
C     contains only cosine terms.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(I) = COS ((I - 1.0) * 2.0 * PI / (N - 1.0))
  100 CONTINUE
C
      CALL COSTI (N, WSAVE)
      PRINT 1000
      PRINT 1010, (X(I), I = 1, N)
      CALL COST (N, X, WSAVE)
      PRINT 1020
      PRINT 1010, (X(I), I = 1, N)
      CALL COST (N, X, WSAVE)
      PRINT 1030
      PRINT 1010, (X(I), I = 1, N)
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT (1X, 100(F5.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
      END
my_system% f95 -dalign fft_ex10.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   1.0   0.7   0.0  -0.7  -1.0  -0.7   0.0   0.7   1.0
 Transformed Sequence:
   0.0   0.0   8.0   0.0   0.0   0.0   0.0   0.0   0.0
 Recovered Sequence:
  16.0  11.3   0.0 -11.3 -16.0 -11.3   0.0  11.3  16.0
```

SINQ: Sine Quarter-Wave Routines

The following routines are used to compute a a discrete Fourier sine transform or inverse transform of a of a sine series that contains only odd wave numbers.

Routine	Function
`[D]SINQI`	Initialize work array `WSAVE` for `[D]SINQF` or `[D]SINQB`
`[D]SINQF`	Compute Fourier coefficients of sine series with only odd wave numbers
`[D]SINQB`	Compute periodic sequence from Fourier coefficients
`V[D]SINQI`	Initialize work array for `V[D]SINQF` or `V[D]SINQB`
`V[D]SINQF`	Compute Fourier coefficients of multiple sine series with only odd wave numbers
`V[D]SINQB`	Compute multiple periodic sequences from Fourier coefficients

Because of the assumption of symmetry, the sequence used as input to the sine quarter-wave routine only needs to contain the part of the sequence that is sufficient to determine the entire sequence.

Normalization

The *x*SINQ operations are unnormalized inverses of themselves, so a call to *x*SINQF followed by a call to *x*SINQB will multiply the input sequence by $4 \times N$. The V*x*SINQ operations are normalized, so a call of V*x*SINQF followed by a call of V*x*SINQB will return the original sequence.

Sample Programs: Sine Quarter-Wave Transform and Inverse Transform

CODE EXAMPLE 5-11 on page 103 uses `SINQF` to compute sine quarter-wave transform of a real sequence and `SINQB` to compute the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is unnormalized and can be normalized by dividing each value by `4*N`.

CODE EXAMPLE 5-11 Sine Quarter-Wave Transform and Inverse Transform

```
my_system% cat fft_ex11.f
      PROGRAM TEST
      INTEGER            N
      PARAMETER         (N = 6)
      INTEGER            I
      REAL               PI, WSAVE(3 * N + 15), X(N)
      EXTERNAL           SINQB, SINQF, SINQI
      INTRINSIC          ACOS, SIN
C
C     Initialize array X to a real odd quarter-wave sequence,
C     that is, it can be expanded in terms of a sine series with
C     only odd wave number.
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(I) = 40.0 * SIN (I * PI / (2.0 * N))
  100 CONTINUE
C
      PRINT 1000
      PRINT 1010, (X(I), I = 1, N)
      CALL SINQI (N, WSAVE)
      CALL SINQF (N, X, WSAVE)
      PRINT 1020
      PRINT 1010, (X(I), I = 1, N)
      CALL SINQB(N, X, WSAVE)
      PRINT 1030
      PRINT 1010, (X(I), I = 1, N)
C
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT (1X, 100(F6.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
C
      END
my_system% f95 -dalign fft_ex11.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   10.4   20.0   28.3   34.6   38.6   40.0
 Transformed Sequence:
  240.0    0.0    0.0    0.0    0.0    0.0
 Recovered Sequence:
  248.5  480.0  678.8  831.4  927.3  960.0
```

CODE EXAMPLE 5-12 uses VSINQF to compute the sine quarter-wave transform of a single real sequence and VSINQB to compute the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is normalized.

CODE EXAMPLE 5-12 Sine Quarter-Wave Transform and Inverse Transform Using Vector Routines

```
my_system% cat fft_ex12.f
      PROGRAM TEST
C
      INTEGER          M, N
      PARAMETER       (M = 1)
      PARAMETER       (N = 6)
      INTEGER          I
      REAL             PI, WSAVE(N + 15), X(M, N), XT(M, N)
      EXTERNAL         VSINQB, VSINQF, VSINQI
      INTRINSIC        ACOS, SIN
C
C     Initialize the first row of the array to a real odd
C     quarter-wave sequence, that is, it can be expanded in
C     terms of a cosine series with only odd wave numbers.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(M,I) = 40.0 * SIN ((I * PI / (2.0 * N)))
  100 CONTINUE
C
      CALL VSINQI (N, WSAVE)
      PRINT 1000
      PRINT 1010, (X(M, I), I = 1, N)
      CALL VSINQF (M, N, X, XT, M, WSAVE)
      PRINT 1020
      PRINT 1010, (X(M, I), I = 1, N)
      CALL VSINQB (M, N, X, XT, M, WSAVE)
      PRINT 1030
      PRINT 1010, (X(M, I), I = 1, N)
C
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT (1X, 100(F5.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
      END
```

CODE EXAMPLE 5-12 Sine Quarter-Wave Transform and Inverse Transform Using Vector Routines *(Continued)*

```
my_system% f95 -dalign fft_ex12.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
  10.4  20.0  28.3  34.6  38.6  40.0
 Transformed Sequence:
  49.0   0.0   0.0   0.0   0.0   0.0
 Recovered Sequence:
  10.4  20.0  28.3  34.6  38.6  40.0
```

CODE EXAMPLE 5-13 uses VSINQF to compute the sine quarter-wave transform of multiple real sequences and VSINQB to compute the inverse transforms. The computed Fourier coefficients of each sequence are packed and stored in the rows of the original real array. The inverse transforms are normalized.

CODE EXAMPLE 5-13 Sine Quarter-Wave Transform and Inverse Transform Using Vector Routines

```
my_system% cat fft_ex13.f
      PROGRAM TEST
      INTEGER           M, N
      PARAMETER        (M = 4)
      PARAMETER        (N = 6)
      INTEGER           I, J
     REAL              PI, WSAVE(N + 15), X(M, N+1), XT(M, N + 1)
C
      EXTERNAL          VSINQB, VSINQF, VSINQI
      INTRINSIC         ACOS, SIN
C
C     Initialize the array to m real odd quarter-wave sequence,
C     that is, they can be expanded in terms of a cosine series
C     with only odd wave numbers.
C
      PI = ACOS (-1.0)
      DO 110, J=1, M
        DO 100, I=1, N
          X(J,I) = 40.0 * J * SIN (I * PI / (2.0 * N))
  100   CONTINUE
  110 CONTINUE
C
```

CODE EXAMPLE 5-13 Sine Quarter-Wave Transform and Inverse Transform Using Vector Routines *(Continued)*

```
      CALL VSINQI (N, WSAVE)
      PRINT 1000
      DO 120, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
  120 CONTINUE
      CALL VSINQF (M, N, X, XT, M, WSAVE)
      PRINT 1020
      DO 130, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
  130 CONTINUE
      CALL VSINQB (M, N, X, XT, M, WSAVE)
      PRINT 1030
      DO 140, J=1, M
        PRINT 1010, J, (X(J, I), I = 1, N)
  140 CONTINUE
C
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT (1X, '  Sequence', I2, ':  ', 100(F5.1, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
C
      END
my_system% f95 -dalign fft_ex13.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
   Sequence 1:   10.4  20.0  28.3  34.6  38.6  40.0
   Sequence 2:   20.7  40.0  56.6  69.3  77.3  80.0
   Sequence 3:   31.1  60.0  84.9 103.9 115.9 120.0
   Sequence 4:   41.4  80.0 113.1 138.6 154.5 160.0
 Transformed Sequence:
   Sequence 1:   49.0   0.0   0.0   0.0   0.0   0.0
   Sequence 2:   98.0   0.0   0.0   0.0   0.0   0.0
   Sequence 3:  147.0   0.0   0.0   0.0   0.0   0.0
   Sequence 4:  196.0   0.0   0.0   0.0   0.0   0.0
 Recovered Sequence:
   Sequence 1:   10.4  20.0  28.3  34.6  38.6  40.0
   Sequence 2:   20.7  40.0  56.6  69.3  77.3  80.0
   Sequence 3:   31.1  60.0  84.9 103.9 115.9 120.0
   Sequence 4:   41.4  80.0 113.1 138.6 154.5 160.0
```

SINT: Sine Odd-Wave Transform Routines

The following routines are used to perform a discrete Fourier sine transform of an odd sequence.

Routine	Function
`[D]SINTI`	Initialize work array `WSAVE` for `[D]SINQF` or `[D]SINQB`
`[D]SINT`	Compute the Fourier coefficients or inverse transform of a sine series with only odd wave numbers
`V[D]SINTI`	Initialize work array for `V[D]SINQF` or `V[D]SINQB`
`V[D]SINT`	Compute the Fourier coefficients or inverse transform of multiple sine series with only odd wave numbers

The sine odd-wave routines are their own inverse. *x*SINT computes the Fourier coefficients from a periodic sequence or the periodic sequence from the Fourier coefficients. *x*SINTF and *x*SINTB routines do not exist for sine odd-wave transforms.

Because of the assumption of symmetry, the sequence used as input to the sine odd-wave routine only needs to contain the part of the sequence that is sufficient to determine the whole sequence.

Normalization

The *x*SINT transforms are unnormalized inverses of themselves, so a call of *x*SINT followed by another call of *x*SINT will multiply the input sequence by $2 \times (N+1)$. The V*x*SINT transforms are normalized, so a call of V*x*SINT followed by a call of V*x*SINT will return the original sequence.

Sample Program: Sine Odd-Wave Transform

CODE EXAMPLE 5-14 on page 108 uses `SINT` to compute the sine odd-wave transform of a real sequence and the inverse transform. The computed Fourier coefficients are packed and stored in the original real array. The inverse transform is unnormalized and can be normalized by dividing each value by `2*(N+1)`.

CODE EXAMPLE 5-14 Sine Odd-Wave Transform and Inverse Transform

```
my_system% cat fft_ex14.f
      PROGRAM TEST
      INTEGER             N
      PARAMETER          (N = 9)
      INTEGER             I
      REAL                PI, WSAVE(3 * N + 15), X(N)
      EXTERNAL            SINT, SINTI
      INTRINSIC           ACOS, SIN
C
C     Initialize the array X to an odd sequence, that is, it
C     can be expanded in terms of a trigonometric series that
C     contains only sine terms.
C
      PI = ACOS (-1.0)
      DO 100, I=1, N
        X(I) =  SIN ( I * 2.0 * PI / (N + 1.0))
  100 CONTINUE
C
      PRINT 1000
      PRINT 1010, (X(I), I = 1, N)
      CALL SINTI (N, WSAVE)
      CALL SINT (N, X, WSAVE)
      PRINT 1020
      PRINT 1010, (X(I), I = 1, N)
      CALL SINT (N, X, WSAVE)
      PRINT 1030
      PRINT 1010, (X(I), I = 1, N)
 1000 FORMAT (1X, 'Original Sequence: ')
 1010 FORMAT (1X, 100(F7.3, 1X))
 1020 FORMAT (1X, 'Transformed Sequence: ')
 1030 FORMAT (1X, 'Recovered Sequence: ')
      END
my_system% f95 -dalign fft_ex14.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence:
0.588 0.951 0.951 0.588 0.000 -0.588 -0.951 -0.951 -0.588
 Transformed Sequence:
0.000 10.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
 Recovered Sequence:
11.756 19.021 19.021 11.756 0.000 -11.756 -19.021 -19.021 -11.756
```

Two-Dimensional FFT and Inverse Transform Routines

The following routines are used to compute a two-dimensional fast Fourier transform or inverse transform of a two-dimensional periodic sequence.

Routine	Function
`[R,D,C,Z]FFT2I`	Initialize the work array `WORK` for `[R,D,C,Z]FFT2F` or `[R,D,C,Z]FFT2B`
`[R,D,C,Z]FFT2F`	Compute Fourier coefficients of two-dimensional periodic sequence
`[R,D,C,Z]FFT2B`	Compute periodic sequence from Fourier coefficients

The two-dimensional fast Fourier transform and inverse transform are computed using the following formulas.

`[R,D,C,Z]FFT2F`

$$H(n_1, n_2) = \sum_{k_1 = 0}^{n_2 - 1} \sum_{k_2 = 0}^{n_1 - 1} h(k_1, k_2) \times W_1 \times W_2$$

`[R,D,C,Z]FFT2B`

$$F(n_1, n_2) = (n_1 \times n_2) \sum_{k_1 = 0}^{n_2 - 1} \sum_{k_2 = 0}^{n_1 - 1} f(k_1, k_2) \times W_1 \times W_2$$

n_1, k_1 range from 0 to N_1-1

n_2, k_2 range from 0 to N_2-1

$W_1 = e^{isign2\pi ik_1n_1/N_1}$

$W_2 = e^{isign2\pi ik_2n_2/N_2}$

$i = \sqrt{-1}$

isign = -1 in `[R,D,C,Z]FFT2F`

= 1 in `[R,D,C,Z]FFT2B`

The *x*FFT2F routines compute the two-dimensional FFT by doing the following:

1. Perform a one-dimensional transform of the columns of the input vector.
2. Transpose the result matrix.
3. Perform a one-dimensional transform of the columns of the result matrix.
4. Transpose the result matrix to restore the original order of the data points.

Arguments for Two-Dimensional FFT Routines

Complex two-dimensional FFT routines use the arguments shown in TABLE 5-13.

TABLE 5-13 Arguments for Complex Two-Dimensional FFT Routines

Argument	Definition
`M`	Number of rows to be transformed
`N`	Number of columns to be transformed
`A`	Two-dimensional array `A(LDA,N)` containing the sequences to be transformed and the results of an in-place transform
`LDA`	Leading dimension of array containing data to be transformed
`WORK`	Work array initialized by *x*FFT2I
`LWORK`	Dimension of work array `WORK`

Arguments for `PLACE`, `FULL`, `B`, and `LDB` are not used with the complex two-dimensional FFT routines, because the transformed sequence is stored in the original input array without any additional manipulations.

Real two-dimensional FFT routines use the arguments shown in TABLE 5-14.

TABLE 5-14 Arguments for Real Two-Dimensional FFT Routines

Argument	Definition
`PLACE`	`'I'` or `'i'` specifies that an in-place transform is performed. `'O'` or `'o'` specifies that an out-of-place transform is performed.
`FULL`	`RFFT2F` or `DFFT2F` only: `'F'` or `'f'` specifies that a full result matrix is generated. Any other character specifies that a partial result matrix is generated.
`M`	Number of rows to be transformed

TABLE 5-14 Arguments for Real Two-Dimensional FFT Routines *(Continued)*

Argument	Definition
N	Number of columns to be transformed
A	Two-dimensional array A(LDA,N) containing the sequences to be transformed and the results of an in-place transform
LDA	Leading dimension of array containing data to be transformed
B	Two-dimensional array B(2*LDB,N) that stores the results of an out-of-place transform
LDB	One half of the actual leading dimension of array that stores results of out-of-place transform
WORK	Work array initialized by *x*FFT2I
LWORK	Dimension of work array WORK

Normalization

The *x*FFT2 operations are unnormalized, so a call of *x*FFT2F followed by a call of *x*FFT2B will multiply the input sequence by M*N.

Data Storage for Two-Dimensional FFT Routines

The data storage format for the computed Fourier coefficients depends upon whether the sequence is complex or real.

Storage of Complex Two-Dimensional Sequences

When CFFT2F or ZFFT2F computes the two-dimensional FFT of a complex sequence, all Fourier coefficients are retained, and the results are stored in the original array. Additional storage options for complex two-dimensional sequences are not needed.

Storage of Real Two-Dimensional Sequences

The result of using RFFT2F or DFFT2F to compute the two-dimensional FFT of a real sequence is a complex vector that contains twice the number of values as the input sequence.

The data storage format of real two-dimensional FFT routines depends upon the following storage options.

- **In-place or Out-of-place.** When using In-Place, the results are stored in the modified input array that contains one or two additional rows, depending upon whether `M` is odd or even. When using Out-of-Place, the results are stored in a separate array.
- **Full or Partial.** When using Full, the complex conjugates are retained. When using Partial, the complex conjugates are discarded.

When computing a real one-dimensional FFT, the complex result can be packed and stored in the original array, because the values identically equal to zero and the complex conjugates are not stored. When computing the real two-dimensional FFT using the in-place and partial storage options, the complex conjugates are not stored, but the values identically equal to zero are stored. Saving the values identically equal to zero simplifies the indexing that occurs when computing the two-dimensional FFT. However, the size of the original array is modified to contain one or two additional rows, which are needed to store the values identically equal to zero.

The values of the arguments used with the real two-dimensional FFT routines depend upon whether an in-place or out-of place transform is performed, and whether the results are stored in a full or partial result matrix, as shown in TABLE 5-15.

TABLE 5-15 Relationships Between Values of Arguments for Real Two-Dimensional FFT Routines

	Full Result Matrix	Partial Result Matrix
In-Place Transform	`B` unused	`B` unused
	`LDB` unused	`LDB` unused
	`LDA` must be even	`LDA` must be even
	`LDA` ≥ `2*M`	`LDA` ≥ `M+2` if `M` is even `LDA` ≥ `M+1` if `M` is odd
	`A(1:2*M, 1:N)`	`A(1:M+2, 1:N)` if `M` is even `A(1:M+1, 1:N)` if `M` is odd
Out-of-Place Transform	`A` unchanged	`A` unchanged
	`LDA` ≥ `M`	`LDA` ≥ `M`
	`2*LDB` ≥ `M`	`2*LDB` ≥ `M+2` if `M` is even `2*LDB` ≥ `M+1` if `M` is odd
	`B(1:2*M, 1:N)`	`B(1:M+2, 1:N)` if `M` is even `B(1:M+1, 1:N)` if `M` is odd

When computing the real two-dimensional FFT of an input sequence of `M` rows and `N` columns, the computed Fourier coefficients will be stored in a two-dimensional array with `2*M` rows and `N` columns when using the Full storage option. When using the Partial storage option, the Fourier coefficients will be stored in a two-dimensional array with `M+2` rows and `N` columns when `M` is even, or in a two-dimensional array with `M+1` rows and `N` columns when `M` is odd.

For example, if M=4 and N=2, the Fourier coefficients will be stored in the output array as follows:

```
Full Storage Option

X(1,1) = Re(X_0)  X(1,2) = Re(X_0)
X(2,1) = Im(X_0)  X(2,2) = Im(X_0)
X(3,1) = Re(X_1)  X(3,2) = Re(X_1)
X(4,1) = Im(X_1)  X(4,2) = Im(X_1)
X(5,1) = Re(X_2)  X(5,2) = Re(X_2)
X(6,1) = Im(X_2)  X(6,2) = Im(X_2)
X(7,1) = Re(X_3)  X(7,2) = Re(X_3)
X(8,1) = Im(X_3)  X(8,2) = Im(X_3)

Partial Storage Option
X(1,1) = Re(X_0)  X(1,2) = Re(X_0)
X(2,1) = Im(X_0)  X(2,2) = Im(X_0)
X(3,1) = Re(X_1)  X(3,2) = Re(X_1)
X(4,1) = Im(X_1)  X(4,2) = Im(X_1)
X(5,1) = Re(X_2)  X(5,2) = Re(X_2)
X(6,1) = Im(X_2)  X(6,2) = Im(X_2)
```

Using Two-Dimensional FFT Routines to Perform Two-Dimensional Convolution

Sun Performance Library provides the `[S,D,C,Z]CNVCOR` routines for computing the convolution or correlation of a filter with one or more input vectors and the `[S,D,C,Z]CNVCOR2` routines for computing the two-dimensional convolution or correlation of two matrices. These routines are described in Section "Convolution and Correlation Routines" on page 135.

The two-dimensional FFT routines can also be used to compute the two-dimensional convolutions of the two two-dimensional arrays A and B, as described in the following procedure.

1. **Compute the two-dimensional FFT of A.**
2. **Compute the two-dimensional FFT of B.**
3. **Perform pointwise multiplication of A and B.**
4. **Compute the inverse two-dimensional FFT of the previous result.**

   The second transpose can be avoided for increased performance by using the VFFT and [SDCZ]TRANS routines to explicitly compute the transposed two-dimensional FFT, as described in the following procedure.

1. **Use VFFT to compute one dimensional FFTs along the columns of A.**
2. **Use ZTRANS to transpose A.**
3. **Use VFFT to compute one-dimensional FFTs along the columns of the new A.**
4. **Use VFFT to compute one-dimensional FFTs along the columns of B.**
5. **Use ZTRANS to transpose B.**
6. **Use VFFT to compute one-dimensional FFTs along the columns of the new B.**
7. **Perform pointwise multiplication of A and B.**
8. **Use VFFT to compute inverse one-dimensional FFTs along the columns of the result.**
9. **Use ZTRANS to transpose the result back into its original order.**

Sample Program: Two-Dimensional FFT and Inverse Transform

CODE EXAMPLE 5-15 uses CFFT2F to compute the two-dimensional FFT of a two-dimensional complex sequence and CFFT2B to compute the inverse transform. The computed Fourier coefficients are stored in the original complex array. The inverse transform is unnormalized and can be normalized by dividing each value by M*N.

CODE EXAMPLE 5-15 Two-Dimensional FFT and Inverse of Complex Sequence

```
my_system: cat fft_ex15.f
      PROGRAM TEST
C
      INTEGER            LWORK, M, N
      PARAMETER         (M = 2)
      PARAMETER         (N = 4)
      PARAMETER         (LWORK = 4 * (M + N + N) + 40)
      INTEGER            I, J
      REAL               PI, WORK(LWORK)
      REAL               X, Y
      COMPLEX            A(M,N)
C
      EXTERNAL           CFFT2B, CFFT2F, CFFT2I
      INTRINSIC          ACOS, CMPLX, COS, SIN
C
C     Initialize the array C to a complex sequence.
      PI = ACOS (-1.0)
      DO 110, J = 1, N
        DO 100, I = 1, M
          X = SIN ((I - 1.0) * 2.0 * PI / N)
          Y = COS ((J - 1.0) * 2.0 * PI / M)
          A(I,J) = CMPLX (X, Y)
  100   CONTINUE
  110 CONTINUE
C
      PRINT 1000
      DO 200, I = 1, M
        PRINT 1010, (A(I,J), J = 1, N)
  200 CONTINUE
      CALL CFFT2I (M, N, WORK)
      CALL CFFT2F (M, N, A, M, WORK, LWORK)
      PRINT 1020
```

CODE EXAMPLE 5-15 Two-Dimensional FFT and Inverse of Complex Sequence *(Continued)*

```
      DO 300, I = 1, M
        PRINT 1010, (A(I,J), J = 1, N)
  300 CONTINUE
      CALL CFFT2B (M, N, A, M, WORK, LWORK)
      PRINT 1030
      DO 400, I = 1, M
        PRINT 1010, (A(I,J), J = 1, N)
  400 CONTINUE
C
 1000 FORMAT (1X, 'Original Sequences:')
 1010 FORMAT (1X, 100(F4.1,' +',F4.1,'i  '))
 1020 FORMAT (1X, 'Transformed Sequences:')
 1030 FORMAT (1X, 'Recovered Sequences:')
C
      END
my_system: f95 -dalign fft_ex15.f -xlic_lib=sunperf
my_system: a.out
 Original Sequences:
  0.0 + 1.0i   0.0 +-1.0i   0.0 + 1.0i   0.0 +-1.0i
  1.0 + 1.0i   1.0 +-1.0i   1.0 + 1.0i   1.0 +-1.0i
 Transformed Sequences:
  4.0 + 0.0i   0.0 + 0.0i   0.0 + 8.0i   0.0 + 0.0i
 -4.0 + 0.0i   0.0 + 0.0i   0.0 + 0.0i   0.0 + 0.0i
 Recovered Sequences:
  0.0 + 8.0i   0.0 +-8.0i   0.0 + 8.0i   0.0 +-8.0i
  8.0 + 8.0i   8.0 +-8.0i   8.0 + 8.0i   8.0 +-8.0i
```

CODE EXAMPLE 5-16 on page 117 uses `RFFT2F` to compute the two-dimensional FFT of a real two-dimensional sequence and `RFT2B` to compute the inverse transform. This example uses the `FULL` storage option and PLACE set to `'O'` for out-of-place storage.

The computed Fourier coefficients are stored in a (2*M, N) array where one row contains the real part of the complex coefficient and the next row contains the imaginary part of the complex coefficient. In CODE EXAMPLE 5-15, to better display the complex conjugate symmetry, the real and imaginary parts of each complex coefficient are displayed on one line.

For example, the following output:

```
 Transformed Out-of-Place, Full
  (   6.241,   0.000)  (   1.173,   0.000)
  (  -0.018,   1.169)  (   0.304,   0.111)
```

represents the following values for the Fourier coefficients.

Column 1		Column 2	
Re(X_0)	Im(X_0)	Re(X_0)	Im(X_0)
Re(X_1)	Im(X_1)	Re(X_1)	Im(X_1)

The inverse transform is unnormalized and can be normalized by dividing each value by M*N.

CODE EXAMPLE 5-16 RFFT2F and RFFT2B Example Showing In-Place and Out-of-Place Storage

```
my_system% cat fft_ex16.f
      PROGRAM TESTFFT
      INTEGER M, N
      PARAMETER(M = 6, N = 2)
      CALL FFT(M,N)
      END

      SUBROUTINE FFT(M, N)
      CHARACTER*1 IS_FULL
      INTEGER I, J, M, N, ISTAT, LWORK, LDA, LDB, LDB_ACTUAL
      REAL RNUM, RAND
      EXTERNAL RFFT2F, RFFT2B, RFFT2I, RAND
      REAL, DIMENSION(:,:), ALLOCATABLE :: AT, B, INPUT
      REAL, DIMENSION(:), ALLOCATABLE :: WT
      LDA = 2*M
      LDB = 2*M

      LWORK = M+2*N+MAX(M,2*N)+30
      ALLOCATE(AT(LDA,N), INPUT(LDA,N), WT(LWORK), B(LDB_ACTUAL,N))
```

CODE EXAMPLE 5-16 RFFT2F and RFFT2B Example Showing In-Place and Out-of-Place Storage *(Continued)*

```
      CALL RFFT2I (M, N, WT)
      DO  I = 1, N
        DO  J = 1, M
          INPUT(J,I) = RAND(0)
        END DO
      END DO
      AT = INPUT
*
      PRINT *, 'Original Sequence'
      DO I = 1, M
        PRINT '(100(F8.3))', (AT(I,J), J = 1, N)
      END DO
      PRINT *
*
*     Example 1
*     Out-of-place, full
*     leading dimension of B (2*LDB) must be at least 2*M
*
      IS_FULL = 'F'
      LDB = M
      CALL RFFT2F ('O', IS_FULL, M, N, AT, LDA, B, LDB, WT, LWORK)
      PRINT *, 'Transformed Out-of-Place, Full'
      DO I = 1, LDB_ACTUAL, N
        PRINT '(100(''  ('', F8.3, '','', F8.3, '')'' :))',
     $     (B(I,J), B(I+1,J), J = 1, N)
      END DO
*      B(M+3:LDB,1:N) = 0
*      PRINT *, 'Transformed, last half clear:'
*      DO I = 1, LDB, N
*        PRINT '(100(''  ('', F8.3, '','', F8.3, '')'' :))',
*     $     (B(I,J), B(I+1,J), J = 1, N)
*      END DO
      CALL RFFT2B ('O', M, N, AT, LDA, B, LDB, WT, LWORK)
      PRINT *, 'Inverse: Scaled Output, Out-of-Place, Full'
      DO I = 1, M
        PRINT '(100(F8.3))', (AT(I,J) / (M * N), J = 1, N)
      END DO
      PRINT *
*
```

CODE EXAMPLE 5-16 RFFT2F and RFFT2B Example Showing In-Place and Out-of-Place Storage *(Continued)*

```
*     Example 2
*     in-place, full
*     LDA must be at least 2*M
*
      AT = INPUT
      IS_FULL = 'F'
      CALL RFFT2F ('I', IS_FULL, M, N, AT, LDA, 0, 0, WT, LWORK)
      PRINT *, 'Transformed In-Place, Full'
      DO I = 1, LDA, 2
        PRINT '(100(''  ('', F8.3, '','', F8.3, '')'' :))',
     $     (AT(I,J), AT(I+1,J), J = 1, N)
      END DO
      CALL RFFT2B ('I', M, N, AT, LDA, 0, 0, WT, LWORK)
      PRINT *, 'Inverse: Scaled Output, In-Place, Full'
      DO I = 1, M
        PRINT '(100(F8.3))', (AT(I,J) / (M * N), J = 1, N)
      END DO
      PRINT *
      DEALLOCATE(AT,WT,B)
      END SUBROUTINE
```

CODE EXAMPLE 5-16 RFFT2F and RFFT2B Example Showing In-Place and Out-of-Place Storage *(Continued)*

```
my_system% f95 -dalign fft_ex16.f -xlic_lib=sunperf
my_system% a.out
 Original Sequence
   0.968   0.654
   0.067   0.021
   0.478   0.512
   0.910   0.202
   0.352   0.940
   0.933   0.204

 Transformed Out-of-Place, Full
  (   6.241,   0.000)  (   1.173,   0.000)
  (  -0.018,   1.169)  (   0.304,   0.111)
  (   0.981,   0.647)  (   0.945,   1.071)
  (   1.569,   0.000)  (  -1.790,   0.000)
  (   0.981,  -0.647)  (   0.945,  -1.071)
  (  -0.018,  -1.169)  (   0.304,  -0.111)
 Inverse: Scaled Output, Out-of-Place, Full
   0.968   0.654
   0.067   0.021
   0.478   0.512
   0.910   0.202
   0.352   0.940
   0.933   0.204

 Transformed In-Place, Full
  (   6.241,   0.000)  (   1.173,   0.000)
  (  -0.018,   1.169)  (   0.304,   0.111)
  (   0.981,   0.647)  (   0.945,   1.071)
  (   1.569,   0.000)  (  -1.790,   0.000)
  (   0.981,  -0.647)  (   0.945,  -1.071)
  (  -0.018,  -1.169)  (   0.304,  -0.111)
 Inverse: Scaled Output, In-Place, Full
   0.968   0.654
   0.067   0.021
   0.478   0.512
   0.910   0.202
   0.352   0.940
   0.933   0.204
```

CODE EXAMPLE 5-17 is a C example that uses `zfft2f` to compute the two-dimensional FFT of a two-dimensional complex sequence and `zfft2b` to compute the inverse transform. The computed Fourier coefficients are stored in the original complex array. The inverse transform is unnormalized and can be normalized by dividing each value by `m*n`.

CODE EXAMPLE 5-17 ZFFT2F and ZFFT2B Example Using C

```
my_system% cat fft_ex17.c
#include <sunperf.h>
#include <math.h>
#include <stdlib.h>

/*
 * This code demonstrates the use of zfft2i, zfft2f, zfft2b
 */
void
main()
{
  int                     i,j,ip;
  int                     m,n,max_mn;
  int                     lwork,lda;
  doublecomplex    *a;
  double                *work;
  double                scale;
  double                err,maxerr;

  m = 16; n = 8;
  a = (doublecomplex *)malloc(m*n*sizeof(doublecomplex));
  max_mn = m; if (n > m) max_mn = n;
  lwork = 2*(m+n+max_mn)+40;
  work = (double *)malloc(lwork*sizeof(double));

  /* initialize a as complex(sin(i),sin(j)) */

  ip = 0;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      a[ip].r=sin((double)i);
      a[ip].i=sin((double)j);
      ip++;
    }
  }
```

CODE EXAMPLE 5-17 ZFFT2F and ZFFT2B Example Using C *(Continued)*

```
  zfft2i(m,n,work);

  lda = m;

  /* compute the forward fft */

  zfft2f(m,n,a,lda,(doublecomplex *)&work,lwork);

  /* compute the inverse fft. Note that the same work array can
     be used for both the forward and the inverse fft */

  zfft2b(m,n,a,lda,(doublecomplex *)&work,lwork);

  /* the reconstruction result will be scaled by m*n */

  scale = (double)(m*n);

  maxerr = 0.0;

  ip = 0;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      err = fabs(a[ip].r/scale-sin((double)i))+
      fabs(a[ip].i/scale-sin((double)j));
      if (err > maxerr) maxerr = err;
      ip++;
    }
  }

  printf("reconstruction error %g \n",maxerr);

  /* clean up */
  free(a);
  free(work);
}
```

CODE EXAMPLE 5-18 on page 123 is a C example that uses `rfft2f` to compute the two-dimensional FFT of a two-dimensional real sequence and `rfft2b` to compute the inverse transform. The computed Fourier coefficients are stored in the original real array using the partial storage option. The inverse transform is unnormalized and can be normalized by dividing each value by `m*n`.

CODE EXAMPLE 5-18 Example of Using the Partial Storage Option

```
my_system% fft_ex18.c
#include <sunperf.h>
#include <math.h>
#include <stdlib.h>
/*
 This code demonstrates the use of dfft2i, dfft2f
 a is being initialized as a 2D real array of size
 m x n = 8 x 4:
 a =
   0.700000   1.375463  -0.296165   1.493668
   0.995520   1.127380  -0.225815   1.638000
   1.264642   0.841120  -0.072764   1.698543
   1.483327   0.542254   0.149314   1.669890
   1.632039   0.257480   0.420585   1.554599
   1.697495   0.012234   0.716814   1.362969
   1.673848  -0.171576   1.011541   1.112118
   1.563209  -0.277530   1.278440   0.824454

 The 2D FFT of a is:
 A =
 Columns 0 through 2:
   29.05310 +  0.00000i    8.02813 +  7.64742i   -1.06904 +  0.00000i
   -1.09423 -  0.24829i   -1.78923 -  3.37830i   -2.81937 +  7.27093i
   -0.21980 -  0.09124i   -0.16036 -  1.30903i   -2.62181 +  2.67179i
   -0.08924 -  0.03707i    0.20683 -  0.80372i   -2.59231 +  1.08567i
   -0.06281 +  0.00000i    0.38653 -  0.53453i   -2.58634 +  0.00000i
   -0.08924 +  0.03707i    0.50611 -  0.32973i   -2.59231 -  1.08567i
   -0.21980 +  0.09124i    0.57617 -  0.14256i   -2.62181 -  2.67179i
   -1.09423 +  0.24829i    0.21514 -  0.20391i   -2.81937 -  7.27093i

 Column 3:
    8.02813 -  7.64742i
    0.21514 +  0.20391i
    0.57617 +  0.14256i
    0.50611 +  0.32973i
    0.38653 +  0.53453i
    0.20683 +  0.80372i
   -0.16036 +  1.30903i
   -1.78923 +  3.37830i
```

CODE EXAMPLE 5-18 Example of Using the Partial Storage Option *(Continued)*

```
 To use dfft2f with the 'in-place' and 'partial storage'  options,
 a has to be embedded into an  (m+2) x n = 10 x 8 real array (case
 m even). After calling dfft2f, this array contains the (m/2+1) x n =
 5 x 4 upper half of the complex result (the lower part can be determined
 via the conjugate symmetry property of the result along the first
 dimension.

 The result of dfft2f will be:

   A(0:4,:) =

 Columns 0 through 2:

   29.05310 +  0.00000i    8.02813 +  7.64742i   -1.06904 +  0.00000i
   -1.09423 -  0.24829i   -1.78923 -  3.37830i   -2.81937 +  7.27093i
   -0.21980 -  0.09124i   -0.16036 -  1.30903i   -2.62181 +  2.67179i
   -0.08924 -  0.03707i    0.20683 -  0.80372i   -2.59231 +  1.08567i
   -0.06281 +  0.00000i    0.38653 -  0.53453i   -2.58634 +  0.00000i

 Column 3:

    8.02813 -  7.64742i
    0.21514 +  0.20391i
    0.57617 +  0.14256i
    0.50611 +  0.32973i
    0.38653 +  0.53453i

  This result is stored in the original real array, i.e. a(0,0) contains
  29.05310, a(1,0) contains 0.00000, a(2,0) contains -1.09423 etc.

 */
void
main()
{
  int           i,j,ipa;
  int           ip;
  int           m,n,max_m2n,max_mn;
  int           lwork,lda;
  double        *a;
  double        *work_a;
  char          place,full;
```

CODE EXAMPLE 5-18 Example of Using the Partial Storage Option *(Continued)*

```
  m = 8; n = 4;
  lda = m+2;

  a = (double *)malloc(lda*n*sizeof(double));

  max_m2n = m; if (2*n > m) max_m2n = 2*n;

  lwork = 2*(m+n+max_m2n)+30;

  work_a = (double *)malloc(lwork*sizeof(double));

  /* initialize a */

  ipa = 0;
  ip  = 0;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      a[ipa]=sin(.3*ip)+.7;
      ipa++;
      ip++;
    }
    ipa+=2;
  }

  dfft2i(m,n,work_a);

  full = 'N';
  place = 'I';

  dfft2f(place,full,m,n,a,lda,NULL,0,work_a,lwork);

  /* clean up */
  free );
  free(work_a);
}
```

Three-Dimensional FFT and Inverse Transform Routines

The following routines are used to perform a three-dimensional fast Fourier transform or inverse transform of a three-dimensional periodic sequence.

Routine	Function
`[R,D,C,Z]FFT3I`	Initialize the work array `WORK` for `[R,D,C,Z]FFT3F or [R,D,C,Z]FFT3B`
`[R,D,C,Z]FFT3F`	Compute Fourier coefficients of three-dimensional periodic sequence
`[R,D,C,Z]FFT3B`	Compute periodic sequence from Fourier coefficients

`[R,D,C,Z]FFT3F`

$$H(n_1, n_2, n_3) = \sum_{k_1=0}^{n_3-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_1-1} h(k_1, k_2, k_3) \times W_1 \times W_2 \times W_3$$

`[R,D,C,Z]FFT3B`

$$F(n_1, n_2, n_3) = (n_1 \times n_2 \times n_3) \sum_{k_1=0}^{n_3-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_1-1} f(k_1, k_2, k_3) \times W_1 \times W_2 \times W_3$$

n_1, k_1 range from 0 to N_1-1

n_2, k_2 range from 0 to N_2-1

n_3, k_3 range from 0 to N_3-1

$W_1 = e^{isign2\pi ik_1n_1/N_1}$

$W_2 = e^{isign2\pi ik_2n_2/N_2}$

$W_3 = e^{isign2\pi ik_3n_3/N_3}$

$i = \sqrt{-1}$

isign = -1 in `[R,D,C,Z]FFT3F`

= 1 in `[R,D,C,Z]FFT3B`

The *x*FFT3F routines compute the three-dimensional FFT by doing the following:

1. Perform a one-dimensional transform of the columns of the input vector.
2. Transpose the result matrix.
3. Perform a one-dimensional transform of the columns of the result matrix.
4. Reflect the result matrix so that the planes become columns.
5. Perform a one-dimensional transform of the columns of the result matrix.
6. Reflect and transpose the result matrix to restore the original order of the data points.

Arguments for Three-Dimensional FFT Routines

Complex three-dimensional FFT routines use the arguments shown in TABLE 5-16.

TABLE 5-16 Arguments for Complex Three-Dimensional FFT Routines

Argument	Definition
`M`	Number of rows to be transformed
`N`	Number of columns to be transformed
`K`	Number of planes to be transformed
`A`	Three-dimensional array `A(LDA,N,K)` containing the sequences to be transformed and the results of an in-place transform
`LDA`	Leading dimension of array containing data to be transformed, where `LDA` ≥ `M`
`LD2A`	Second dimension of array to be transformed, where `LD2A` ≥ `N`
`WORK`	Work array initialized by *x*FFT3I
`LWORK`	Dimension of work array `WORK`

Arguments for `PLACE`, `FULL`, `B`, and `LDB` are not used with the complex three-dimensional FFT routines, because the transformed sequence is stored in the original input array without any additional manipulations.

Real three-dimensional FFT routines use the arguments shown in TABLE 5-17.

TABLE 5-17 Arguments for Real Three-Dimensional FFT Routines

Argument	Definition
`PLACE`	`'I'` or `'i'` specifies that an in-place transform is performed. `'O'` or `'o'` specifies that an out-of-place transform is performed.
`FULL`	`RFFT3F` or `DFFT3F` only: `'F'` or `'f'` specifies that a full result matrix is generated. Any other character specifies that a partial result matrix is generated.
`M`	Number of rows to be transformed
`N`	Number of columns to be transformed
`K`	Number of planes to be transformed
`A`	Three-dimensional array `A(LDA,N,K)` containing the sequences to be transformed and the results of an in-place transform
`LDA`	Leading dimension of array containing data to be transformed
`B`	Three-dimensional array `B(2*LDB,N,K)` that stores the results of an out-of-place transform
`LDB`	Leading dimension of array that stores results of out-of-place transform
`WORK`	Work array initialized by *x*FFT3I
`LWORK`	Dimension of work array `WORK`

Normalization

The *x*FFT3 operations are unnormalized, so a call of *x*FFT3F followed by a call of *x*FFT3B will multiply the input sequence by `M*N*K`.

Data Storage for Three-Dimensional FFT Routines

The data storage format for the computed Fourier coefficients depends upon whether the sequence is complex or real.

Storage of Complex Three-Dimensional Sequences

When CFFT3F or ZFFT3F computes the three-dimensional FFT of a complex sequence, all Fourier coefficients are retained, and the results are stored in the original three-dimensional array A(LDA, LD2A, K). Additional storage options for complex three-dimensional sequences are not required.

Storage of Real Three-Dimensional Sequences

The result of using RFFT3F or DFFT3F to compute the three-dimensional FFT of a real sequence is a complex vector that contains twice the number of values as the input sequence.

The data storage format of real three-dimensional FFT routines depends upon the following storage options.

- **In-place or Out-of-place.** When using In-Place, the results are stored in the modified input array that contains one or two additional rows, depending upon whether M is odd or even. When using Out-of-Place, the results are stored in a separate array.
- **Full or Partial.** When using Full, complex conjugates are retained. When using Partial, the complex conjugates are discarded.

When computing a real one-dimensional FFT, the complex result can be packed and stored in the original array, because the values identically equal to zero and the complex conjugates are not stored. When computing the real three-dimensional FFT using the in-place and partial storage options, the complex conjugates are not stored, but the values identically equal to zero are stored. Saving the values identically equal to zero simplifies the indexing that occurs when computing the three-dimensional FFT. However, the size of the original array is modified to contain one or two additional rows, which are needed to store the values identically equal to zero.

The values of the arguments used with the real three-dimensional FFT routines depend upon whether an in-place or out-of place transform is performed, and whether the results are stored in a full or partial result matrix, as shown in TABLE 5-18.

TABLE 5-18 Relationship Between Values of Arguments for Real Three-Dimensional FFT Routines

	Full Result Array	Partial Result Array
In-Place Transform	B unused	B unused
	LDB unused	LDB unused
	LDA must be even	LDA must be even
	LDA ≥ 2*M	LDA ≥ M+2 if M is even LDA ≥ M+1 if M is odd
	A(1:2*M, 1:N)	A(1:M+2, 1:N) if M is even A(1:M+1, 1:N) if M is odd
Out-of-Place Transform	A unchanged	A unchanged
	LDA ≥ M	LDA ≥ M
	LDB ≥ 2*M	LDB ≥ M/2+1 if M is even LDB ≥ (M-1)/2+1 if M is odd
	B(1:2*M, 1:N, 1:K)	B(1:M+2, 1:N, 1:K) if M is even B(1:M+1, 1:N, 1:K) if M is odd

When computing the real 3D FFT of an input sequence of M rows, N columns, and K planes, the computed Fourier coefficients will be stored in a result matrix with 2*M rows, N columns for each value of K when using the Full storage option. When using the Partial storage option, the Fourier coefficients will be stored in a result matrix with M+2 rows and N columns for each value of K when M is even, or in M+1 rows and N columns when M is odd. For each value of K, the storage format of the Fourier coefficients in the M rows and N columns is the same as for the real two-dimensional FFT routines. See "Storage of Real Two-Dimensional Sequences" on page 111.

Sample Program: Three-Dimensional FFT and Inverse Transform

CODE EXAMPLE 5-19 uses `CFFT3F` to compute the three-dimensional FFT of a three-dimensional complex sequence and `CFFT3B` to compute the inverse transform. The computed Fourier coefficients are stored in the original complex array. The inverse transform is unnormalized and can be normalized by dividing each value by `M*N*K`.

CODE EXAMPLE 5-19 Three-Dimensional Fast Fourier Transform and Inverse Transform

```
my_system% cat fft_ex19.f
      PROGRAM TEST
      INTEGER          LWORK, M, N, K
      PARAMETER        (K = 2)
      PARAMETER        (M = 2)
      PARAMETER        (N = 4)
      PARAMETER        (LWORK = 4 * (M + N + N) + 45)
      INTEGER           I, J, L
      REAL              PI, WORK(LWORK)
      REAL              X, Y
      COMPLEX           C(M,N,K)
C
      EXTERNAL          CFFT3B, CFFT3F, CFFT3I
      INTRINSIC         ACOS, CMPLX, COS, SIN
C     Initialize the array C to a complex sequence.
      PI = ACOS (-1.0)
      DO 120, L = 1, K
        DO 110, J = 1, N
          DO 100, I = 1, M
            X = SIN ((I - 1.0) * 2.0 * PI / N)
            Y = COS ((J - 1.0) * 2.0 * PI / M)
            C(I,J,L) = CMPLX (X, Y)
  100     CONTINUE
  110   CONTINUE
  120 CONTINUE
C
```

CODE EXAMPLE 5-19 Three-Dimensional Fast Fourier Transform and Inverse Transform *(Continued)*

```
      PRINT 1000
      DO 210, L = 1, K
        PRINT 1010, L
        DO 200, I = 1, M
          PRINT 1020, (C(I,J,L), J = 1, N)
  200   CONTINUE
  210 CONTINUE
      CALL CFFT3I (M, N, K, WORK)
      CALL CFFT3F (M, N, K, C, M, N, WORK, LWORK)
      PRINT 1030
      DO 310, L = 1, K
        PRINT 1010, L
        DO 300, I = 1, M
          PRINT 1020, (C(I,J,L), J = 1, N)
  300   CONTINUE
  310 CONTINUE
      CALL CFFT3B (M, N, K, C, M, N, WORK, LWORK)
      PRINT 1040
      DO 410, L = 1, K
        PRINT 1010, L
        DO 400, I = 1, M
          PRINT 1020, (C(I,J,L), J = 1, N)
  400   CONTINUE
  410 CONTINUE
C
 1000 FORMAT (1X, 'Original Sequences:')
 1010 FORMAT (1X, '  Plane', I2)
 1020 FORMAT (5X, 100(F5.1,' +',F5.1,'i  '))
 1030 FORMAT (/1X, 'Transformed Sequences:')
 1040 FORMAT (/1X, 'Recovered Sequences:')
      END
```

CODE EXAMPLE 5-19 Three-Dimensional Fast Fourier Transform and Inverse Transform *(Continued)*

```
my_system% f95 -dalign fft_ex19.f -xlic_lib=sunperf
my_system% a.out
 Original Sequences:
   Plane 1
       0.0 +  1.0i    0.0 + -1.0i    0.0 +  1.0i    0.0 + -1.0i
       1.0 +  1.0i    1.0 + -1.0i    1.0 +  1.0i    1.0 + -1.0i
   Plane 2
       0.0 +  1.0i    0.0 + -1.0i    0.0 +  1.0i    0.0 + -1.0i
       1.0 +  1.0i    1.0 + -1.0i    1.0 +  1.0i    1.0 + -1.0i

 Transformed Sequences:
   Plane 1
       8.0 +  0.0i    0.0 +  0.0i    0.0 + 16.0i    0.0 +  0.0i
      -8.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i
   Plane 2
       0.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i
       0.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i    0.0 +  0.0i

 Recovered Sequences:
   Plane 1
       0.0 + 16.0i    0.0 +-16.0i    0.0 + 16.0i    0.0 +-16.0i
      16.0 + 16.0i   16.0 +-16.0i   16.0 + 16.0i   16.0 +-16.0i
   Plane 2
       0.0 + 16.0i    0.0 +-16.0i    0.0 + 16.0i    0.0 +-16.0i
      16.0 + 16.0i   16.0 +-16.0i   16.0 + 16.0i   16.0 +-16.0i
```

References

For additional information on the DFT or FFT, see the following sources.

Briggs, William L., and Henson, Van Emden. *The DFT: An Owner's Manual for the Discrete Fourier Transform*. Philadelphia, PA: SIAM, 1995.

Brigham, E. Oran. *The Fast Fourier Transform and Its Applications*. Upper Saddle River, NJ: Prentice Hall, 1988.

Chu, Eleanor, and George, Alan. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Boca Raton, FL: CRC Press, 2000.

Press, William H., Teukolsky, Saul A., Vetterling, William T., and Flannery, Brian P. *Numerical Recipes in C: The Art of Scientific Computing*. 2 ed. Cambridge, United Kingdom: Cambridge University Press, 1992.

Ramirez, Robert W. *The FFT: Fundamentals and Concepts*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Swartzrauber, Paul N. Vectorizing the FFTs. In Rodrigue, Garry ed. *Parallel Computations*. New York: Academic Press, Inc., 1982.

Strang, Gilbert. *Linear Algebra and Its Applications*. 3 ed. Orlando, FL: Harcourt Brace & Company, 1988.

Van Loan, Charles. *Computational Frameworks for the Fast Fourier Transform*. Philadelphia, PA: SIAM, 1992.

Walker, James S. *Fast Fourier Transforms*. Boca Raton, FL: CRC Press, 1991.

CHAPTER 6

Using Sun Performance Library Convolution and Correlation Routines

Sun Performance Library contains the convolution routines shown in TABLE 6-1.

TABLE 6-1 Convolution and Correlation Routines

Routine	Arguments	Function
`SCNVCOR`, `DCNVCOR`, `CCNVCOR`, `ZCNVCOR`	`CNVCOR,FOUR,NX,X,IFX, INCX,NY,NPRE,M,Y,IFY, INC1Y,INC2Y,NZ,K,Z, IFZ,INC1Z,INC2Z,WORK, LWORK`	Convolution or correlation of two vectors
`SCNVCOR2`, `DCNVCOR2`, `CCNVCOR2`, `ZCNVCOR2`	`CNVCOR,METHOD,TRANSX, SCRATCHX,TRANSY, SCRATCHY,MX,NX,X,LDX, MY,NY,MPRE,NPRE,Y,LDY, MZ,NZ,Z,LDZ,WORKIN, LWORK`	Convolution or correlation of two matrices

Convolution and Correlation Routines

The `[S,D,C,Z]CNVCOR` routines are used to compute the convolution or correlation of a filter with one or more input vectors. The `[S,D,C,Z]CNVCOR2` routines are used to compute the two-dimensional convolution or correlation of two matrices.

Arguments for Convolution and Correlation Routines

The one-dimensional convolution and correlation routines use the arguments shown in TABLE 6-2.

TABLE 6-2 Arguments for One-Dimensional Convolution and Correlation Routines SCNVCOR, DCNVCOR, CCNVCOR, and ZCNVCOR

Argument	Definition
CNVCOR	'V' or 'v' specifies that convolution is computed. 'R' or 'r' specifies that correlation is computed.
FOUR	'T' or 't' specifies that the Fourier transform method is used. 'D' or 'd' specifies that the direct method is used, where the convolution or correlation is computed from the definition of convolution and correlation. (See Note 1)
NX	Length of filter vector, where NX ≥ 0.
X	Filter vector
IFX	Index of first element of X, where NX ≥ IFX ≥ 1
INCX	Stride between elements of the vector in X, where INCX > 0.
NY	Length of input vectors, where NY ≥ 0.
NPRE	Number of implicit zeros prefixed to the Y vectors, where NPRE ≥ 0.
M	Number of input vectors, where M ≥ 0.
Y	Input vectors.
IFY	Index of the first element of Y, where NY ≥ IFY ≥ 1
INC1Y	Stride between elements of the input vectors in Y, where INC1Y > 0.
INC2Y	Stride between input vectors in Y, where INC2Y > 0.
NZ	Length of the output vectors, where NZ ≥ 0.
K	Number of Z vectors, where K ≥ 0. If K < M, only the first K vectors will be processed. If K > M, all input vectors will be processed and the last M-K output vectors will be set to zero on exit.
Z	Result vectors
IFZ	Index of the first element of Z, where NZ ≥ IFZ ≥ 1
INC1Z	Stride between elements of the output vectors in Z, where INCYZ > 0.

TABLE 6-2 Arguments for One-Dimensional Convolution and Correlation Routines SCNVCOR, DCNVCOR, CCNVCOR, and ZCNVCOR *(Continued)*

Argument	Definition
INC2Z	Stride between output vectors in Z, where INC2Z > 0.
WORK	Work array
LWORK	Length of work array

Note 1. When the lengths of the two sequences to be convolved are similar, the FFT method is faster than the direct method. However, when one sequence is much larger than the other, such as when convolving a large time-series signal with a small filter, the direct method performs faster than the FFT-based method.

The two-dimensional convolution and correlation routines use the arguments shown in TABLE 6-3.

TABLE 6-3 Arguments for Two-Dimensional Convolution and Correlation Routines SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2

Argument	Definition
CNVCOR	'V' or 'v' specifies that convolution is computed. 'R' or 'r' specifies that correlation is computed.
METHOD	'T' or 't' specifies that the Fourier transform method is used. 'D' or 'd' specifies that the direct method is used, where the convolution or correlation is computed from the definition of convolution and correlation. (See Note 1)
TRANSX	'N' or 'n' specifies that X is the filter matrix 'T' or 't' specifies that the transpose of X is the filter matrix
SCRATCHX	'N' or 'n' specifies that X must be preserved 'S' or 's' specifies that X can be used for scratch space. The contents of X are undefined after returning from a call where X is used for scratch space.
TRANSY	'N' or 'n' specifies that Y is the input matrix 'T' or 't' specifies that the transpose of Y is the input matrix
SCRATCHY	'N' or 'n' specifies that Y must be preserved 'S' or 's' specifies that Y can be used for scratch space. The contents of X are undefined after returning from a call where Y is used for scratch space.
MX	Number of rows in the filter matrix X, where MX ≥ 0
NX	Number of columns in the filter matrix X, where NX ≥ 0
X	Filter matrix. X is unchanged on exit when SCRATCHX is 'N' or 'n' and undefined on exit when SCRATCHX is 'S' or 's'.

TABLE 6-3 Arguments for Two-Dimensional Convolution and Correlation Routines SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2 *(Continued)*

Argument	Definition
LDX	Leading dimension of array containing the filter matrix X.
MY	Number of rows in the input matrix Y, where MY ≥ 0.
NY	Number of columns in the input matrix Y, where NY ≥ 0
MPRE	Number of implicit zeros prefixed to each row of the input matrix Y vectors, where MPRE ≥ 0.
NPRE	Number of implicit zeros prefixed to each column of the input matrix Y, where NPRE ≥ 0.
Y	Input matrix. Y is unchanged on exit when SCRATCHY is 'N' or 'n' and undefined on exit when SCRATCHY is 'S' or 's'.
LDY	Leading dimension of array containing the input matrix Y.
MZ	Number of output vectors, where MZ ≥ 0.
NZ	Length of output vectors, where NZ ≥ 0.
Z	Result vectors
LDZ	Leading dimension of the array containing the result matrix Z, where LDZ ≥ MAX(1,MZ).
WORKIN	Work array
LWORK	Length of work array

Note 1. When the sizes of the two matrices to be convolved are similar, the FFT method is faster than the direct method. However, when one sequence is much larger than the other, such as when convolving a large data set with a small filter, the direct method performs faster than the FFT-based method.

Work Array WORK for Convolution and Correlation Routines

The minimum dimensions for the WORK work arrays used with the one-dimensional and two-dimensional convolution and correlation routines are shown in TABLE 6-6 on page 140. The minimum dimensions for one-dimensional convolution and correlation routines depend upon the values of the arguments NPRE, NX, NY, and NZ.

The minimum dimensions for two-dimensional convolution and correlation routines depend upon the values of the arguments shown TABLE 6-4.

TABLE 6-4 Arguments Affecting Minimum Work Array Size for Two-Dimensional Routines: SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2

Argument	Definition
MX	Number of rows in the filter matrix
MY	Number of rows in the input matrix
MZ	Number of output vectors
NX	Number of columns in the filter matrix
NY	Number of columns in the input matrix
NZ	Length of output vectors
MPRE	Number of implicit zeros prefixed to each row of the input matrix
NPRE	Number of implicit zeros prefixed to each column of the input matrix
MPOST	MAX(0,MZ-MYC)
NPOST	MAX(0,NZ-NYC)
MYC	MPRE + MPOST + MYC_INIT, where MYC_INIT depends upon filter and input matrices, as shown in TABLE 6-5
NYC	NPRE + NPOST + NYC_INIT, where NYC_INIT depends upon filter and input matrices, as shown in TABLE 6-5

MYC_INIT and NYC_INIT depend upon the following, where X is the filter matrix and Y is the input matrix.

TABLE 6-5 MYC_INIT and NYC_INIT Dependencies

	Y		Transpose(Y)	
	X	Transpose(X)	X	Transpose(X)
MYC_INIT	MAX(MX,MY)	MAX(NX,MY)	MAX(MX,NY)	MAX(NX,NY)
NYC_INIT	MAX(NX,NY)	MAX(MX,NY)	MAX(NX,MY)	MAX(MX,MY)

The values assigned to the minimum work array size is shown in TABLE 6-6.

TABLE 6-6 Minimum Dimensions and Data Types for `WORK` Work array Used With Convolution and Correlation Routines

Routine	Minimum Work Array Size (`WORK`)	Type
SCNVCOR, DCNVCOR	4*(MAX(NX,NPRE+NY) + MAX(0,NZ-NY))	REAL, REAL*8
CCNVCOR, ZCNVCOR	2*(MAX(NX,NPRE+NY) + MAX(0,NZ-NY)))	COMPLEX, COMPLEX*16
SCNVCOR2[1], DCNVCOR2[1]	MY + NY + 30	COMPLEX, COMPLEX*16
CCNVCOR2[1], ZCNVCOR2[1]	If MY = NY: MYC + 8 If MY ≠ NY: MYC + NYC + 16	COMPLEX, COMPLEX*16

1. Memory will be allocated within the routine if the workspace size, indicated by LWORK, is not large enough.

Sample Program: Convolution

CODE EXAMPLE 6-1 uses `CCNVCOR` to perform FFT convolution of two complex vectors.

CODE EXAMPLE 6-1 One-Dimensional Convolution Using Fourier Transform Method and `COMPLEX` Data

```
my_system% cat con_ex20.f
      PROGRAM TEST
C
      INTEGER           LWORK
      INTEGER           N
      PARAMETER        (N = 3)
      PARAMETER        (LWORK = 4 * N + 15)
      COMPLEX           P1(N), P2(N), P3(2*N-1), WORK(LWORK)
      DATA P1 / 1, 2, 3 /,  P2 / 4, 5, 6 /
C
      EXTERNAL          CCNVCOR
C
      PRINT *, 'P1:'
      PRINT 1000, P1
      PRINT *, 'P2:'
      PRINT 1000, P2
```

CODE EXAMPLE 6-1 One-Dimensional Convolution Using Fourier Transform Method and COMPLEX Data *(Continued)*

```
      CALL CCNVCOR ('V', 'T', N, P1, 1, 1, N, 0, 1, P2, 1, 1, 1,
     $              2 * N - 1, 1, P3, 1, 1, 1, WORK, LWORK)
C
      PRINT *, 'P3:'
      PRINT 1000, P3
C
 1000 FORMAT (1X, 100(F4.1,' +',F4.1,'i  '))
C
      END
my_system% f95 -dalign con_ex20.f -xlic_lib=sunperf
my_system% a.out
 P1:
  1.0 + 0.0i   2.0 + 0.0i   3.0 + 0.0i
 P2:
  4.0 + 0.0i   5.0 + 0.0i   6.0 + 0.0i
 P3:
  4.0 + 0.0i  13.0 + 0.0i  28.0 + 0.0i  27.0 + 0.0i  18.0 + 0.0i
```

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 can 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.

Convolution can be used to compute the products of polynomials. CODE EXAMPLE 6-2 uses SCNVCOR to compute the product of $1 + 2x + 3x^2$ and $4 + 5x + 6x^2$.

CODE EXAMPLE 6-2 One-Dimensional Convolution Using Fourier Transform Method and REAL Data

```
my_system% cat con_ex21.f
      PROGRAM TEST
      INTEGER     LWORK, NX, NY, NZ
      PARAMETER  (NX = 3)
      PARAMETER  (NY = NX)
      PARAMETER  (NZ = 2*NY-1)
      PARAMETER  (LWORK = 4*NZ+32)
      REAL        X(NX), Y(NY), Z(NZ), WORK(LWORK)
C
      DATA X / 1, 2, 3 /,  Y / 4, 5, 6 /, WORK / LWORK*0 /
C
      PRINT 1000, ’X’
      PRINT 1010, X
      PRINT 1000, ’Y’
      PRINT 1010, Y
      CALL SCNVCOR (’V’, ’T’, NX, X, 1, 1,
     $NY, 0, 1, Y, 1, 1, 1,   NZ, 1, Z, 1, 1, 1, WORK, LWORK)
      PRINT 1020, ’Z’
      PRINT 1010, Z
 1000 FORMAT (1X, ’Input vector ’, A1)
 1010 FORMAT (1X, 300F5.0)
 1020 FORMAT (1X, ’Output vector ’, A1)
      END
my_system% f95 -dalign con_ex21.f -xlic_lib=sunperf
my_system% a.out
 Input vector X
    1.   2.   3.
 Input vector Y
    4.   5.   6.
 Output vector Z
    4.  13.  28.  27.  18.
```

Making the output vector longer than the input vectors, as in the example above, implicitly adds zeros to the end of the input. No zeros are actually required in any of the vectors, and none are used in the example, but the padding provided by the implied zeros has the effect of an end-off shift rather than an end-around shift of the input vectors.

CODE EXAMPLE 6-3 will compute the product between the vector [1, 2, 3] and the circulant matrix defined by the initial column vector [4, 5, 6]:

CODE EXAMPLE 6-3 Convolution Used to Compute the Product of a Vector and Circulant Matrix

```
my_system% cat con_ex22.f
      PROGRAM TEST
C
      INTEGER     LWORK, NX, NY, NZ
      PARAMETER  (NX = 3)
      PARAMETER  (NY = NX)
      PARAMETER  (NZ = NY)
      PARAMETER  (LWORK = 4*NZ+32)
      REAL        X(NX), Y(NY), Z(NZ), WORK(LWORK)
C
      DATA X / 1, 2, 3 /,  Y / 4, 5, 6 /, WORK / LWORK*0 /
C
      PRINT 1000, ’X’
      PRINT 1010, X
      PRINT 1000, ’Y’
      PRINT 1010, Y
      CALL SCNVCOR (’V’, ’T’, NX, X, 1, 1,
     $NY, 0, 1, Y, 1, 1, 1,   NZ, 1, Z, 1, 1, 1,
     $WORK, LWORK)
      PRINT 1020, ’Z’
      PRINT 1010, Z
C
 1000 FORMAT (1X, ’Input vector ’, A1)
 1010 FORMAT (1X, 300F5.0)
 1020 FORMAT (1X, ’Output vector ’, A1)
      END
my_system% f95 -dalign con_ex22.f -xlic_lib=sunperf
my_system% a.out
 Input vector X
    1.   2.   3.
 Input vector Y
    4.   5.   6.
 Output vector Z
   31.  31.  28.
```

The difference between this example and the previous example is that the length of the output vector is the same as the length of the input vectors, so there are no implied zeros on the end of the input vectors. With no implied zeros to shift into, the effect of an end-off shift from the previous example does not occur and the end-around shift results in a circulant matrix product.

CODE EXAMPLE 6-4 Two-Dimensional Convolution Using Direct Method

```
my_system% cat con_ex23.f
      PROGRAM TEST
C
      INTEGER           M, N
      PARAMETER        (M = 2)
      PARAMETER        (N = 3)
C
      INTEGER           I, J
      COMPLEX           P1(M,N), P2(M,N), P3(M,N)
      DATA P1 / 1, -2, 3, -4, 5, -6 /,  P2 / -1, 2, -3, 4, -5, 6 /
      EXTERNAL          CCNVCOR2
C
      PRINT *, 'P1:'
      PRINT 1000, ((P1(I,J), J = 1, N), I = 1, M)
      PRINT *, 'P2:'
      PRINT 1000, ((P2(I,J), J = 1, N), I = 1, M)
C
      CALL CCNVCOR2 ('V', 'Direct', 'No Transpose X', 'No Overwrite X',
     $   'No Transpose Y', 'No Overwrite Y', M, N, P1, M,
     $   M, N, 0, 0, P2, M, M, N, P3, M, 0, 0)
C
      PRINT *, 'P3:'
      PRINT 1000, ((P3(I,J), J = 1, N), I = 1, M)
C
 1000 FORMAT (3(F5.1,' +',F5.1,'i  '))
C
      END
```

CODE EXAMPLE 6-4 Two-Dimensional Convolution Using Direct Method *(Continued)*

```
my_system% f95 -dalign con_ex23.f -xlic_lib=sunperf
my_system% a.out
 P1:
  1.0 +  0.0i    3.0 +  0.0i    5.0 +  0.0i
 -2.0 +  0.0i   -4.0 +  0.0i   -6.0 +  0.0i
 P2:
 -1.0 +  0.0i   -3.0 +  0.0i   -5.0 +  0.0i
  2.0 +  0.0i    4.0 +  0.0i    6.0 +  0.0i
 P3:
-83.0 +  0.0i  -83.0 +  0.0i  -59.0 +  0.0i
 80.0 +  0.0i   80.0 +  0.0i   56.0 +  0.0i
```

APPENDIX **A**

Sun Performance Library Routines

This appendix lists the Sun Performance Library routines by library, routine name, and function.

For a description of the function and a listing of the Fortran and C interfaces, refer to the section 3P man pages for the individual routines. For example, to display the man page for the `SBDSQR` routine, type **`man -s 3P sbdsqr`**. The man page routine names use lowercase letters.

For many routines, separate routines exist that operate on different data types. Rather than list each routine separately, a lowercase *x* is used in a routine name to denote single, double, complex, and double complex data types. For example, the routine *x*`BDSQR` is available as four routines that operate with the following data types:

- `SBDSQR` – Single data type
- `BBDSQR` – Double data type
- `CBDSQR` – Complex data type
- `ZBDSQR` – Double complex data type

If a routine name is not available for `S`, `B`, `C`, and `Z`, the *x* prefix will not be used and each routine name will be listed.

LAPACK Routines

TABLE A-1 LAPACK (Linear Algebra Package) Routines

Routine	Function
Bidiagonal Matrix	
SBDSDC or DBDSDC	Computes the singular value decomposition (SVD) of a bidirectional matrix, using a divide and conquer method.
*x*BDSQR	Computes SVD of real upper or lower bidiagonal matrix, using the bidirectional QR algorithm.
Diagonal Matrix	
SDISNA or DDISNA	Computes the reciprocal condition numbers for eigenvectors of real symmetric or complex Hermitian matrix.
General Band Matrix	
*x*GBBRD	Reduces real or complex general band matrix to upper bidiagonal form.
*x*GBCON	Estimates the reciprocal of the condition number of general band matrix using LU factorization.
*x*GBEQU	Computes row and column scalings to equilibrate a general band matrix and reduce its condition number.
*x*GBRFS	Refines solution to general banded system of linear equations.
*x*GBSV	Solves a general banded system of linear equations (simple driver).
*x*GBSVX	Solves a general banded system of linear equations (expert driver).
*x*GBTRF	LU factorization of a general band matrix using partial pivoting with row interchanges.
*x*GBTRS	Solves a general banded system of linear equations, using the factorization computed by *x*GBTRF.
General Matrix (Unsymmetric or Rectangular)	
*x*GEBAK	Forms the right or left eigenvectors of a general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by *x*GEBAL.
*x*GEBAL	Balances a general matrix.
*x*GEBRD	Reduces a general matrix to upper or lower bidiagonal form by an orthogonal transformation.
*x*GECON	Estimates the reciprocal of the condition number of a general matrix, using the factorization computed by *x*GETRF.
*x*GEEQU	Computes row and column scalings intended to equilibrate a general rectangular matrix and reduce its condition number.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
*x*GEES	Computes the eigenvalues and Schur factorization of a general matrix (simple driver).
*x*GEESX	Computes the eigenvalues and Schur factorization of a general matrix (expert driver).
*x*GEEV	Computes the eigenvalues and left and right eigenvectors of a general matrix (simple driver).
*x*GEEVX	Computes the eigenvalues and left and right eigenvectors of a general matrix (expert driver).
*x*GEGS	Depreciated routine replaced by *x*GGES.
*x*GEGV	Depreciated routine replaced by *x*GGEV.
*x*GEHRD	Reduces a general matrix to upper Hessenberg form by an orthogonal similarity transformation.
*x*GELQF	Computes LQ factorization of a general rectangular matrix.
*x*GELS	Computes the least squares solution to an over-determined system of linear equations using a QR or LQ factorization of A.
*x*GELSD	Computes the least squares solution to an over-determined system of linear equations using a divide and conquer method using a QR or LQ factorization of A.
*x*GELSS	Computes the minimum-norm solution to a linear least squares problem by using the SVD of a general rectangular matrix (simple driver).
*x*GELSX	Depreciated routine replaced by *x*SELSY.
*x*GELSY	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization.
*x*GEQLF	Computes QL factorization of a general rectangular matrix.
*x*GEQP3	Computes QR factorization of general rectangular matrix using Level 3 BLAS.
*x*GEQPF	Depreciated routine replaced by *x*GEQP3.
*x*GEQRF	Computes QR factorization of a general rectangular matrix.
*x*GERFS	Refines solution to a system of linear equations.
*x*GERQF	Computes RQ factorization of a general rectangular matrix.
*x*GESDD	Computes SVD of general rectangular matrix using a divide and conquer method.
*x*GESV	Solves a general system of linear equations (simple driver).
*x*GESVX	Solves a general system of linear equations (expert driver).
*x*GESVD	Computes SVD of general rectangular matrix.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
*x*GETRF	Computes an LU factorization of a general rectangular matrix using partial pivoting with row interchanges.
*x*GETRI	Computes inverse of a general matrix using the factorization computed by *x*GETRF.
*x*GETRS	Solves a general system of linear equations using the factorization computed by *x*GETRF.
General Matrix-Generalized Problem (Pair of General Matrices)	
*x*GGBAK	Forms the right or left eigenvectors of a generalized eigenvalue problem based on the output by *x*GGBAL.
*x*GGBAL	Balances a pair of general matrices for the generalized eigenvalue problem.
*x*GGES	Computes the generalized eigenvalues, Schur form, and left and/or right Schur vectors for two nonsymmetric matrices.
*x*GGESX	Computes the generalized eigenvalues, Schur form, and left and/or right Schur vectors.
*x*GGEV	Computes the generalized eigenvalues and the left and/or right generalized eigenvalues for two nonsymmetric matrices.
*x*GGEVX	Computes the generalized eigenvalues and the left and/or right generalized eigenvectors.
*x*GGGLM	Solves the GLM (Generalized Linear Regression Model) using the GQR (Generalized QR) factorization.
*x*GGHRD	Reduces two matrices to generalized upper Hessenberg form using orthogonal transformations.
*x*GGLSE	Solves the LSE (Constrained Linear Least Squares Problem) using the GRQ (Generalized RQ) factorization.
*x*GGQRF	Computes generalized QR factorization of two matrices.
*x*GGRQF	Computes generalized RQ factorization of two matrices.
*x*GGSVD	Computes the generalized singular value decomposition.
*x*GGSVP	Computes an orthogonal or unitary matrix as a preprocessing step for calculating the generalized singular value decomposition.
General Tridiagonal Matrix	
*x*GTCON	Estimates the reciprocal of the condition number of a tridiagonal matrix, using the LU factorization as computed by *x*GTTRF.
*x*GTRFS	Refines solution to a general tridiagonal system of linear equations.
*x*GTSV	Solves a general tridiagonal system of linear equations (simple driver).
*x*GTSVX	Solves a general tridiagonal system of linear equations (expert driver).

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
*x*GTTRF	Computes an LU factorization of a general tridiagonal matrix using partial pivoting and row exchanges.
*x*GTTRS	Solves general tridiagonal system of linear equations using the factorization computed by *x*.
Hermitian Band Matrix	
CHBEV or ZHBEV	(Replacement with newer version CHBEVD or ZHBEVD suggested) Computes all eigenvalues and eigenvectors of a Hermitian band matrix.
CHBEVD or ZHBEVD	Computes all eigenvalues and eigenvectors of a Hermitian band matrix and uses a divide and conquer method to calculate eigenvectors.
CHBEVX or ZHBEVX	Computes selected eigenvalues and eigenvectors of a Hermitian band matrix.
CHBGST or ZHBGST	Reduces Hermitian-definite banded generalized eigenproblem to standard form.
CHBGV or ZHBGV	(Replacement with newer version CHBGVD or ZHBGVD suggested) Computes all eigenvalues and eigenvectors of a generalized Hermitian-definite banded eigenproblem.
CHBGVD or ZHBGVD	Computes all eigenvalues and eigenvectors of generalized Hermitian-definite banded eigenproblem and uses a divide and conquer method to calculate eigenvectors.
CHBGVX or ZHBGVX	Computes selected eigenvalues and eigenvectors of a generalized Hermitian-definite banded eigenproblem.
CHBTRD or ZHBTRD	Reduces Hermitian band matrix to real symmetric tridiagonal form by using a unitary similarity transform.
Hermitian Matrix	
CHECON or ZHECON	Estimates the reciprocal of the condition number of a Hermitian matrix using the factorization computed by CHETRF or ZHETRF.
CHEEV or ZHEEV	(Replacement with newer version CHEEVR or ZHEEVR suggested) Computes all eigenvalues and eigenvectors of a Hermitian matrix (simple driver).
CHEEVD or ZHEEVD	(Replacement with newer version CHEEVR or ZHEEVR suggested) Computes all eigenvalues and eigenvectors of a Hermitian matrix and uses a divide and conquer method to calculate eigenvectors.
CHEEVR or ZHEEVR	Computes selected eigenvalues and the eigenvectors of a complex Hermitian matrix.
CHEEVX or ZHEEVX	Computes selected eigenvalues and eigenvectors of a Hermitian matrix (expert driver).
CHEGST or ZHEGST	Reduces a Hermitian-definite generalized eigenproblem to standard form using the factorization computed by CPOTRF or ZPOTRF.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
CHEGV or ZHEGV	(Replacement with newer version CHEGVD or ZHEGVD suggested) Computes all the eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem.
CHEGVD or ZHEGVD	Computes all the eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem and uses a divide and conquer method to calculate eigenvectors.
CHEGVX or ZHEGVX	Computes selected eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem.
CHERFS or ZHERFS	Improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite.
CHESV or ZHESV	Solves a complex Hermitian indefinite system of linear equations (simple driver).
CHESVX or ZHESVX	Solves a complex Hermitian indefinite system of linear equations (simple driver).
CHETRD or ZHETRD	Reduces a Hermitian matrix to real symmetric tridiagonal form by using a unitary similarity transformation.
CHETRF or ZHERTF	Computes the factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
CHETRI or ZHETRI	Computes the inverse of a complex Hermitian indefinite matrix, using the factorization computed by CHETRF or ZHETRF.
CHETRS or ZHETRS	Solves a complex Hermitian indefinite matrix, using the factorization computed by CHETRF or ZHETRF.
Hermitian Matrix in Packed Storage	
CHPCON or ZHPCON	Estimates the reciprocal of the condition number of a Hermitian indefinite matrix in packed storage using the factorization computed by CHPTRF or ZHPTRF.
CHPEV or ZHPEV	(Replacement with newer version CHPEVD or ZHPEVD suggested) Computes all the eigenvalues and eigenvectors of a Hermitian matrix in packed storage (simple driver).
CHPEVX or ZHPEVX	Computes selected eigenvalues and eigenvectors of a Hermitian matrix in packed storage (expert driver).
CHPEVD or ZHPEVD	Computes all the eigenvalues and eigenvectors of a Hermitian matrix in packed storage and uses a divide and conquer method to calculate eigenvectors.
CHPGST or ZHPGST	Reduces a Hermitian-definite generalized eigenproblem to standard form where the coefficient matrices are in packed storage and uses the factorization computed by CPPTRF or ZPPTRF.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
CHPGV or ZHPGV	(Replacement with newer version CHPGVD or ZHPGVD suggested) Computes all the eigenvalues and eigenvectors of a generalized Hermitian-definite eigenproblem where the coefficient matrices are in packed storage (simple driver).
CHPGVX or ZHPGVX	Computes selected eigenvalues and eigenvectors of a generalized Hermitian-definite eigenproblem where the coefficient matrices are in packed storage (expert driver).
CHPGVD or ZHPGVD	Computes all the eigenvalues and eigenvectors of a generalized Hermitian-definite eigenproblem where the coefficient matrices are in packed storage, and uses a divide and conquer method to calculate eigenvectors.
CHPRFS or ZHPRFS	Improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite in packed storage.
CHPSV or ZHPSV	Computes the solution to a complex system of linear equations where the coefficient matrix is Hermitian in packed storage (simple driver).
CHPSVX or ZHPSVX	Uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations where the coefficient matrix is Hermitian in packed storage (expert driver).
CHPTRD or ZHPTRD	Reduces a complex Hermitian matrix stored in packed form to real symmetric tridiagonal form.
CHPTRF or ZHPTRF	Computes the factorization of a complex Hermitian indefinite matrix in packed storage, using the diagonal pivoting method.
CHPTRI or ZHPTRI	Computes the inverse of a complex Hermitian indefinite matrix in packed storage using the factorization computed by CHPTRF or ZHPTRF.
CHPTRS or ZHPTRS	Solves a complex Hermitian indefinite matrix in packed storage, using the factorization computed by CHPTRF or ZHPTRF.
Upper Hessenberg Matrix	
*x*HSEIN	Computes right and/or left eigenvectors of upper Hessenberg matrix using inverse iteration.
*x*HSEQR	Computes eigenvectors and Shur factorization of upper Hessenberg matrix using multishift QR algorithm.
Upper Hessenberg Matrix-Generalized Problem (Hessenberg and Triangular Matrix)	
*x*HGEQZ	Implements single-/double-shift version of QZ method for finding the generalized eigenvalues of the equation det(A - w(i) * B) = 0.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
Real Orthogonal Matrix in Packed Storage	
SOPGTR or DOPGTR	Generates an orthogonal transformation matrix from a tridiagonal matrix determined by SSPTRD or DSPTRD.
SOPMTR or DOPMTR	Multiplies a general matrix by the orthogonal transformation matrix reduced to tridiagonal form by SSPTRD or DSPTRD.
Real Orthogonal Matrix	
SORGBR or DORGBR	Generates the orthogonal transformation matrices from reduction to bidiagonal form, as determined by SGEBRD or DGEBRD.
SORGHR or DORGHR	Generates the orthogonal transformation matrix reduced to Hessenberg form, as determined by SGEHRD or DGEHRD.
SORGLQ or DORGLQ	Generates an orthogonal matrix Q from an LQ factorization, as returned by SGELQF or DGELQF.
SORGQL or DORGQL	Generates an orthogonal matrix Q from a QL factorization, as returned by SGEQLF or DGEQLF.
SORGQR or DORGQR	Generates an orthogonal matrix Q from a QR factorization, as returned by SGEQRF or DGEQRF.
SORGRQ or DORGRQ	Generates orthogonal matrix Q from an RQ factorization, as returned by SGERQF or DGERQF.
SORGTR or DORGTR	Generates an orthogonal matrix reduced to tridiagonal form by SSYTRD or DSYTRD.
SORMBR or DORMBR	Multiplies a general matrix with the orthogonal matrix reduced to bidiagonal form, as determined by SGEBRD or DGEBRD.
SORMHR or DORMHR	Multiplies a general matrix by the orthogonal matrix reduced to Hessenberg form by SGEHRD or DGEHRD.
SORMLQ or DORMLQ	Multiplies a general matrix by the orthogonal matrix from an LQ factorization, as returned by SGELQF or DGELQF.
SORMQL or DORMQL	Multiplies a general matrix by the orthogonal matrix from a QL factorization, as returned by SGEQLF or DGEQLF.
SORMQR or DORMQR	Multiplies a general matrix by the orthogonal matrix from a QR factorization, as returned by SGEQRF or DGEQRF.
SORMR3 or DORMR3	Multiplies a general matrix by the orthogonal matrix returned by STZRZF or DTZRZF.
SORMRQ or DORMRQ	Multiplies a general matrix by the orthogonal matrix from an RQ factorization returned by SGERQF or DGERQF.
SORMRZ or DORMRZ	Multiplies a general matrix by the orthogonal matrix from an RZ factorization, as returned by STZRZF or DTZRZF.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
SORMTR or DORMTR	Multiplies a general matrix by the orthogonal transformation matrix reduced to tridiagonal form by SSYTRD or DSYTRD.
Symmetric or Hermitian Positive Definite Band Matrix	
*x*PBCON	Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite band matrix, using the Cholesky factorization returned by *x*PBTRF.
*x*PBEQU	Computes equilibration scale factors for a symmetric or Hermitian positive definite band matrix.
*x*PBRFS	Refines solution to a symmetric or Hermitian positive definite banded system of linear equations.
*x*PBSTF	Computes a split Cholesky factorization of a real symmetric positive definite band matrix.
*x*PBSV	Solves a symmetric or Hermitian positive definite banded system of linear equations (simple driver).
*x*PBSVX	Solves a symmetric or Hermitian positive definite banded system of linear equations (expert driver).
*x*PBTRF	Computes Cholesky factorization of a symmetric or Hermitian positive definite band matrix.
*x*PBTRS	Solves symmetric positive definite banded matrix, using the Cholesky factorization computed by *x*PBTRF.
Symmetric or Hermitian Positive Definite Matrix	
*x*POCON	Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite matrix, using the Cholesky factorization returned by *x*POTRF.
*x*POEQU	Computes equilibration scale factors for a symmetric or Hermitian positive definite matrix.
*x*PORFS	Refines solution to a linear system in a Cholesky-factored symmetric or Hermitian positive definite matrix.
*x*POSV	Solves a symmetric or Hermitian positive definite system of linear equations (simple driver).
*x*POSVX	Solves a symmetric or Hermitian positive definite system of linear equations (expert driver).
*x*POTRF	Computes Cholesky factorization of a symmetric or Hermitian positive definite matrix.
*x*POTRI	Computes the inverse of a symmetric or Hermitian positive definite matrix using the Cholesky-factorization returned by *x*POTRF.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
*x*POTRS	Solves a symmetric or Hermitian positive definite system of linear equations, using the Cholesky factorization returned by *x*POTRF.
Symmetric or Hermitian Positive Definite Matrix in Packed Storage	
*x*PPCON	Reciprocal condition number of a Cholesky-factored symmetric positive definite matrix in packed storage.
*x*PPEQU	Computes equilibration scale factors for a symmetric or Hermitian positive definite matrix in packed storage.
*x*PPRFS	Refines solution to a linear system in a Cholesky-factored symmetric or Hermitian positive definite matrix in packed storage.
*x*PPSV	Solves a linear system in a symmetric or Hermitian positive definite matrix in packed storage (simple driver).
*x*PPSVX	Solves a linear system in a symmetric or Hermitian positive definite matrix in packed storage (expert driver).
*x*PPTRF	Computes Cholesky factorization of a symmetric or Hermitian positive definite matrix in packed storage.
*x*PPTRI	Computes the inverse of a symmetric or Hermitian positive definite matrix in packed storage using the Cholesky-factorization returned by *x*PPTRF.
*x*PPTRS	Solves a symmetric or Hermitian positive definite system of linear equations where the coefficient matrix is in packed storage, using the Cholesky factorization returned by *x*PPTRF.
Symmetric or Hermitian Positive Definite Tridiagonal Matrix	
*x*PTCON	Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite tridiagonal matrix using the Cholesky factorization returned by *x*PTTRF.
*x*PTEQR	Computes all eigenvectors and eigenvalues of a real symmetric or Hermitian positive definite system of linear equations.
*x*PTRFS	Refines solution to a symmetric or Hermitian positive definite tridiagonal system of linear equations.
*x*PTSV	Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations (simple driver).
*x*PTSVX	Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations (expert driver).
*x*PTTRF	Computes the LDL^H factorization of a symmetric or Hermitian positive definite tridiagonal matrix.
*x*PTTRS	Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations using the LDL^H factorization returned by *x*PTTRF.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
Real Symmetric Band Matrix	
SSBEV or DSBEV	(Replacement with newer version SSBEVD or DSBEVD suggested) Computes all eigenvalues and eigenvectors of a symmetric band matrix.
SSBEVD or DSBEVD	Computes all eigenvalues and eigenvectors of a symmetric band matrix and uses a divide and conquer method to calculate eigenvectors.
SSBEVX or DSBEVX	Computes selected eigenvalues and eigenvectors of a symmetric band matrix.
SSBGST or DSBGST	Reduces symmetric-definite banded generalized eigenproblem to standard form.
SSBGV or DSBGV	(Replacement with newer version SSBGVD or DSBGVD suggested) Computes all eigenvalues and eigenvectors of a generalized symmetric-definite banded eigenproblem.
SSBGVD or DSBGVD	Computes all eigenvalues and eigenvectors of generalized symmetric-definite banded eigenproblem and uses a divide and conquer method to calculate eigenvectors.
SSBGVX or DSBGVX	Computes selected eigenvalues and eigenvectors of a generalized symmetric-definite banded eigenproblem.
SSBTRD or DSBTRD	Reduces symmetric band matrix to real symmetric tridiagonal form by using an orthogonal similarity transform.
Symmetric Matrix in Packed Storage	
*x*SPCON	Estimates the reciprocal of the condition number of a symmetric packed matrix using the factorization computed by xSPTRF.
SSPEV or DSPEV	(Replacement with newer version SSPEVD or DSPEVD suggested) Computes all the eigenvalues and eigenvectors of a symmetric matrix in packed storage (simple driver).
SSPEVX or DSPEVX	Computes selected eigenvalues and eigenvectors of a symmetric matrix in packed storage (expert driver).
SSPEVD or DSPEVD	Computes all the eigenvalues and eigenvectors of a symmetric matrix in packed storage and uses a divide and conquer method to calculate eigenvectors.
SSPGST or DSPGST	Reduces a real symmetric-definite generalized eigenproblem to standard form where the coefficient matrices are in packed storage and uses the factorization computed by SPPTRF or DPPTRF.
SSPGVD or DSPGVD	Computes all the eigenvalues and eigenvectors of a real generalized symmetric-definite eigenproblem where the coefficient matrices are in packed storage, and uses a divide and conquer method to calculate eigenvectors.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
SSPGV or DSPGV	(Replacement with newer version SSPGVD or DSPGVD suggested) Computes all the eigenvalues and eigenvectors of a real generalized symmetric-definite eigenproblem where the coefficient matrices are in packed storage (simple driver).
SSPGVX or DSPGVX	Computes selected eigenvalues and eigenvectors of a real generalized symmetric-definite eigenproblem where the coefficient matrices are in packed storage (expert driver).
*x*SPRFS	Improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite in packed storage.
*x*SPSV	Computes the solution to a system of linear equations where the coefficient matrix is a symmetric matrix in packed storage (simple driver).
*x*SPSVX	Uses the diagonal pivoting factorization to compute the solution to a system of linear equations where the coefficient matrix is a symmetric matrix in packed storage (expert driver).
SSPTRD or DSPTRD	Reduces a real symmetric matrix stored in packed form to real symmetric tridiagonal form using an orthogonal similarity transform.
*x*SPTRF	Computes the factorization of a symmetric packed matrix using the Bunch-Kaufman diagonal pivoting method.
*x*SPTRI	Computes the inverse of a symmetric indefinite matrix in packed storage using the factorization computed by xSPTRF.
*x*SPTRS	Solves a system of linear equations by the symmetric matrix stored in packed format using the factorization computed by xSPTRF.
Real Symmetric Tridiagonal Matrix	
SSTEBZ or DSTEBZ	Computes the eigenvalues of a real symmetric tridiagonal matrix.
*x*STEDC	Computes all the eigenvalues and eigenvectors of a symmetric tridiagonal matrix using a divide and conquer method.
*x*STEGR	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using Relatively Robust Representations.
*x*STEIN	Computes selected eigenvectors of a real symmetric tridiagonal matrix using inverse iteration.
*x*STEQR	Computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using the implicit QL or QR algorithm.
SSTERF or DSTERF	Computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using a root-free QL or QR algorithm variant.
SSTEV or DSTEV	(Replacement with newer version SSTEVR or DSTEVR suggested) Computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix (simple driver).

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
SSTEVX or DSTEVX	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix (expert driver).
SSTEVD or DSTEVD	(Replacement with newer version SSTEVR or DSTEVR suggested) Computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using a divide and conquer method.
SSTEVR or DSTEVR	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using Relatively Robust Representations.
*x*STSV	Computes the solution to a system of linear equations where the coefficient matrix is a symmetric tridiagonal matrix.
*x*STTRF	Computes the factorization of a symmetric tridiagonal matrix.
*x*STTRS	Computes the solution to a system of linear equations where the coefficient matrix is a symmetric tridiagonal matrix.
Symmetric Matrix	
*x*SYCON	Estimates the reciprocal of the condition number of a symmetric matrix using the factorization computed by SSYTRF or DSYTRF.
SSYEV or DSYEV	(Replacement with newer version SSYEVR or DSYEVR suggested) Computes all eigenvalues and eigenvectors of a symmetric matrix.
SSYEVX or DSYEVX	Computes eigenvalues and eigenvectors of a symmetric matrix (expert driver).
SSYEVD or DSYEVD	(Replacement with newer version SSYEVR or DSYEVR suggested) Computes all eigenvalues and eigenvectors of a symmetric matrix and uses a divide and conquer method to calculate eigenvectors.
SSYEVR or DSYEVR	Computes selected eigenvalues and eigenvectors of a symmetric tridiagonal matrix.
SSYGST or DSYGST	Reduces a symmetric-definite generalized eigenproblem to standard form using the factorization computed by SPOTRF or DPOTRF.
SSYGV or DSYGV	(Replacement with newer version SSYGVD or DSYGVD suggested) Computes all the eigenvalues and eigenvectors of a generalized symmetric-definite eigenproblem.
SSYGVX or DSYGVX	Computes selected eigenvalues and eigenvectors of a generalized symmetric-definite eigenproblem.
SSYGVD or DSYGVD	Computes all the eigenvalues and eigenvectors of a generalized symmetric-definite eigenproblem and uses a divide and conquer method to calculate eigenvectors.
*x*SYRFS	Improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite.
*x*SYSV	Solves a real symmetric indefinite system of linear equations (simple driver).

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
*x*SYSVX	Solves a real symmetric indefinite system of linear equations (expert driver).
SSYTRD or DSYTRD	Reduces a symmetric matrix to real symmetric tridiagonal form by using a orthogonal similarity transformation.
*x*SYTRF	Computes the factorization of a real symmetric indefinite matrix using the diagonal pivoting method.
*x*SYTRI	Computes the inverse of a symmetric indefinite matrix using the factorization computed by *x*SYTRF.
*x*SYTRS	Solves a system of linear equations by the symmetric matrix using the factorization computed by *x*SYTRF.
Triangular Band Matrix	
*x*TBCON	Estimates the reciprocal condition number of a triangular band matrix.
*x*TBRFS	Determines error bounds and estimates for solving a triangular banded system of linear equations.
*x*TBTRS	Solves a triangular banded system of linear equations.
Triangular Matrix-Generalized Problem (Pair of Triangular Matrices)	
*x*TGEVC	Computes right and/or left generalized eigenvectors of two upper triangular matrices.
*x*TGEXC	Reorders the generalized Schur decomposition of a real or complex matrix pair using an orthogonal or unitary equivalence transformation.
*x*TGSEN	Reorders the generalized real-Schur or Schur decomposition of two matrixes and computes the generalized eigenvalues.
*x*TGSJA	Computes the generalized SVD from two upper triangular matrices obtained from *x*GGSVP.
*x*TGSNA	Estimates reciprocal condition numbers for specified eigenvalues and eigenvectors of two matrices in real-Schur or Schur canonical form.
*x*TGSYL	Solves the generalized Sylvester equation.
Triangular Matrix in Packed Storage	
*x*TPCON	Estimates the reciprocal or the condition number of a triangular matrix in packed storage.
*x*TPRFS	Determines error bounds and estimates for solving a triangular system of linear equations where the coefficient matrix is in packed storage.
*x*TPTRI	Computes the inverse of a triangular matrix in packed storage.
*x*TPTRS	Solves a triangular system of linear equations where the coefficient matrix is in packed storage.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
Triangular Matrix	
*x*TRCON	Estimates the reciprocal or the condition number of a triangular matrix.
*x*TREVC	Computes right and/or left eigenvectors of an upper triangular matrix.
*x*TREXC	Reorders Schur factorization of matrix using an orthogonal or unitary similarity transformation.
*x*TRRFS	Determines error bounds and estimates for triangular system of a linear equations.
*x*TRSEN	Reorders Schur factorization of matrix to group selected cluster of eigenvalues 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.
*x*TRSNA	Estimates the reciprocal condition numbers of selected eigenvalues and eigenvectors of an upper quasi-triangular matrix.
*x*TRSYL	Solves Sylvester matrix equation.
*x*TRTRI	Computes the inverse of a triangular matrix.
*x*TRTRS	Solves a triangular system of linear equations.
Trapezoidal Matrix	
*x*TZRQF	Depreciated routine replaced by routine *x*TZRZF.
*x*TZRZF	Reduces a rectangular upper trapezoidal matrix to upper triangular form by means of orthogonal transformations.
Unitary Matrix	
CUNGBR or ZUNGBR	Generates the unitary transformation matrices from reduction to bidiagonal form, as determined by CGEBRD or ZGEBRD.
CUNGHR or ZUNGHR	Generates the orthogonal transformation matrix reduced to Hessenberg form, as determined by CGEHRD or ZGEHRD.
CUNGLQ or ZUNGLQ	Generates a unitary matrix Q from an LQ factorization, as returned by CGELQF or ZGELQF.
CUNGQL or ZUNGQL	Generates a unitary matrix Q from a QL factorization, as returned by CGEQLF or ZGEQLF.
CUNGQR or ZUNGQR	Generates a unitary matrix Q from a QR factorization, as returned by CGEQRF or ZGEQRF.
CUNGRQ or ZUNGRQ	Generates a unitary matrix Q from an RQ factorization, as returned by CGERQF or ZGERQF.
CUNGTR or ZUNGTR	Generates a unitary matrix reduced to tridiagonal form, by CHETRD or ZHETRD.

TABLE A-1 LAPACK (Linear Algebra Package) Routines *(Continued)*

Routine	Function
CUNMBR or ZUNMBR	Multiplies a general matrix with the unitary transformation matrix reduced to bidiagonal form, as determined by CGEBRD or ZGEBRD.
CUNMHR or ZUNMHR	Multiplies a general matrix by the unitary matrix reduced to Hessenberg form by CGEHRD or ZGEHRD.
CUNMLQ or ZUNMLQ	Multiplies a general matrix by the unitary matrix from an LQ factorization, as returned by CGELQF or ZGELQF.
CUNMQL or ZUNMQL	Multiplies a general matrix by the unitary matrix from a QL factorization, as returned by CGEQLF or ZGEQLF.
CUNMQR or ZUNMQR	Multiplies a general matrix by the unitary matrix from a QR factorization, as returned by CGEQRF or ZGEQRF.
CUNMRQ or ZUNMRQ	Multiplies a general matrix by the unitary matrix from an RQ factorization, as returned by CGERQF or ZGERQF.
CUNMRZ or ZUNMRZ	Multiplies a general matrix by the unitary matrix from an RZ factorization, as returned by CTZRZF or ZTZRZF.
CUNMTR or ZUNMTR	Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHETRD or ZHETRD.
Unitary Matrix in Packed Storage	
CUPGTR or ZUPGTR	Generates the unitary transformation matrix from a tridiagonal matrix determined by CHPTRD or ZHPTRD.
CUPMTR or ZUPMTR	Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHPTRD or ZHPTRD.

BLAS1 Routines

TABLE A-2 BLAS1 (Basic Linear Algebra Subprograms, Level 1) Routines

Routine	Function
SASUM, DASUM, SCASUM, DZASUM	Sum of the absolute values of a vector
xAXPY	Product of a scalar and vector plus a vector
xCOPY	Copy a vector
SDOT, DDOT, DSDOT, SDSDOT, CDOTU, ZDOTU, DQDOTA, DQDOTI	Dot product (inner product)

TABLE A-2 BLAS1 (Basic Linear Algebra Subprograms, Level 1) Routines *(Continued)*

Routine	Function
`CDOTC, ZDOTC`	Dot product conjugating first vector
`SNRM2, DNRM2, SCNRM2, DCNRM2, DZNRM2`	Euclidean norm of a vector
`xROTG`	Set up Givens plane rotation
`xROT, CSROT, ZDROT`	Apply Given's plane rotation
`SROTMG, DROTMG`	Set up modified Given's plane rotation
`SROTM, DROTM`	Apply modified Given's rotation
`ISAMAX, DAMAX, ICAMAX, IZAMAX`	Index of element with maximum absolute value
`xSCAL, CSSCAL, ZDSCAL`	Scale a vector
`xSWAP`	Swap two vectors
`CVMUL, ZVMUL`	Compute scaled product of complex vectors

BLAS2 Routines

TABLE A-3 BLAS2 (Basic Linear Algebra Subprograms, Level 2) Routines

Routine	Function
x`GBMV`	Product of a matrix in banded storage and a vector
x`GEMV`	Product of a general matrix and a vector
`SGER, DGER, CGERC, ZGERC, CGERU, ZGERU`	Rank-1 update to a general matrix
`CHBMV, ZHBMV`	Product of a Hermitian matrix in banded storage and a vector
`CHEMV, ZHEMV`	Product of a Hermitian matrix and a vector
`CHER, ZHER`	Rank-1 update to a Hermitian matrix
`CHER2, ZHER2`	Rank-2 update to a Hermitian matrix
`CHPMV, ZHPMV`	Product of a Hermitian matrix in packed storage and a vector

TABLE A-3 BLAS2 (Basic Linear Algebra Subprograms, Level 2) Routines *(Continued)*

Routine	Function
`CHPR, ZHPR`	Rank-1 update to a Hermitian matrix in packed storage
`CHPR2, ZHPR2`	Rank-2 update to a Hermitian matrix in packed storage
`SSBMV, DSBMV`	Product of a symmetric matrix in banded storage and a vector
x`SPMV`	Product of a Symmetric matrix in packed storage and a vector
`SSPR, DSPR`	Rank-1 update to a real symmetric matrix in packed storage
`SSPR2, DSPR2`	Rank-2 update to a real symmetric matrix in packed storage
`SSYMV, DSYMV`	Product of a symmetric matrix and a vector
`SSYR, DSYR`	Rank-1 update to a real symmetric matrix
`SSYR2, DSYR2`	Rank-2 update to a real symmetric matrix
x`TBMV`	Product of a triangular matrix in banded storage and a vector
x`TBSV`	Solution to a triangular system in banded storage of linear equations
x`TPMV`	Product of a triangular matrix in packed storage and a vector
x`TPSV`	Solution to a triangular system of linear equations in packed storage
x`TRMV`	Product of a triangular matrix and a vector
x`TRSV`	Solution to a triangular system of linear equations

BLAS3 Routines

TABLE A-4 BLAS3 (Basic Linear Algebra Subprograms, Level 3) Routines

Routine	Function
x`GEMM`	Product of two general matrices
`CHEMM` or `ZHEMM`	Product of a Hermitian matrix and a general matrix
`CHERK` or `ZHERK`	Rank-k update of a Hermitian matrix
`CHER2K` or `ZHER2K`	Rank-2k update of a Hermitian matrix
x`SYMM`	Product of a symmetric matrix and a general matrix

TABLE A-4 BLAS3 (Basic Linear Algebra Subprograms, Level 3) Routines *(Continued)*

Routine	Function
*x*SYRK	Rank-k update of a symmetric matrix
*x*SYR2K	Rank-2k update of a symmetric matrix
*x*TRMM	Product of a triangular matrix and a general matrix
*x*TRSM	Solution for a triangular system of equations

Sparse BLAS Routines

TABLE A-5 Sparse BLAS Routines

Routines	Function
*x*AXPYI	Adds a scalar multiple of a sparse vector *X* to a full vector *Y*.
SBCOMM or DBCOMM	Block coordinate matrix-matrix multiply.
SBDIMM or DBDIMM	Block diagonal format matrix-matrix multiply.
SBDISM or DBDISM	Block Diagonal format triangular solve.
SBELMM or DBELMM	Block Ellpack format matrix-matrix multiply.
SBELSM or DBELSM	Block Ellpack format triangular solve.
SBSCMM or DBSCMM	Block compressed sparse column format matrix-matrix multiply.
SBSCSM or DBSCSM	Block compressed sparse column format triangular solve.
SBSRMM or DBSRMM	Block compressed sparse row format matrix-matrix multiply.
SBSRSM or DBSRSM	Block compressed sparse row format triangular solve.
SCOOMM or DCOOMM	Coordinate format matrix-matrix multiply.
SCSCMM or DCSCMM	Compressed sparse column format matrix-matrix multiply
SCSCSM or DCSCSM	Compressed sparse column format triangular solve

TABLE A-5 Sparse BLAS Routines *(Continued)*

Routines	Function
SCSRMM or DCSRMM	Compressed sparse row format matrix-matrix multiply.
SCSRSM or DCSRSM	Compressed sparse row format triangular solve.
SDIAMM or DDIAMM	Diagonal format matrix-matrix multiply.
SDIASM or DDIASM	Diagonal format triangular solve.
SDOTI, DDOTI, CDOTUI, or ZDOTUI	Computes the dot product of a sparse vector and a full vector.
CDOTCI, or ZDOTCI,	Computes the conjugate dot product of a sparse vector and a full vector.
SELLMM or DELLMM	Ellpack format matrix-matrix multiply.
SELLSM or DELLSM	Ellpack format triangular solve.
*x*CGTHR	Given a full vector, creates a sparse vector and corresponding index vector.
*x*CGTHRZ	Given a full vector, creates a sparse vector and corresponding index vector and zeros the full vector.
SJADMM or DJADMM	Jagged diagonal matrix-matrix multiply.
SJADRP or DJADRP	Right permutation of a jagged diagonal matrix.
SJADSM or DJADSM	Jagged diagonal triangular solve.
SROTI or DROTI	Applies a Givens rotation to a sparse vector and a full vector.
*x*CSCTR	Given a sparse vector and corresponding index vector, puts those elements into a full vector.
SSKYMM or DSKYMM	Skyline format matrix-matrix multiply.

TABLE A-5 Sparse BLAS Routines *(Continued)*

Routines	Function
SSKYSM or DSKYSM	Skyline format triangular solve.
SVBRMM or DVBRMM	Variable block sparse row format matrix-matrix multiply.
SVBRSM or DVBRSM	Variable block sparse row format triangular solve.

Sparse Solver Routines

TABLE A-6 Sparse Solver Routines

Routines	Function
DGSSFS	One call interface to sparse solver.
DGSSIN	Sparse solver initialization.
DGSSOR	Fill reducing ordering and symbolic factorization.
DGSSFA	Matrix value input and numeric factorization.
DGSSSL	Triangular solve.
DGSSUO	Sets user-specified ordering permutation.
DGSSRP	Returns permutation used by solver.
DGSSCO	Returns condition number estimate of coefficient matrix.
DGSSDA	De-allocates sparse solver.
DGSSPS	Prints solver statistics.

FFTPACK and VFFTPACK Routines

Routines with a V prefix are vectorized routines that belong to VFFTPACK.

TABLE A-7 FFTPACK and VFFTPACK (Fast Fourier Transform and Vectorized Fast Fourier Transform) Routines

Routine	Function
`COSQB, DCOSQB, VCOSQB, VDCOSQB`	Cosine quarter-wave synthesis
`COSQF, DCOSQF, VCOSQF, VDCOSQF`	Cosine quarter-wave transform
`COSQI, DCOSQI, VCOSQI, VDCOSQI`	Initialize cosine quarter-wave transform and synthesis
`COST, DCOST, VCOST, VDCOST`	Cosine even-wave transform
`COSTI, DCOSTI, VCOSTI, VDCOSTI`	Initialize cosine even-wave transform
`EZFFTB`	EZ Fourier synthesis
`EZFFTF`	EZ Fourier transform
`EZFFTI`	Initialize EZ Fourier transform and synthesis
`RFFTB, DFFTB, CFFTB, ZFFTB, VRFFTB, VDFFTB, VCFFTB, VZFFTB`	Fourier synthesis
`RFFTF, DFFTF, CFFTF, ZFFTF, VRFFTF, VDFFTF, VCFFTF, VZFFTF`	Fourier transform
`RFFTI, DFFTI, CFFTI, ZFFTI, VRFFTI, VDFFTI, VCFFTI, VZFFTI`	Initialize Fourier transform and synthesis
`SINQB, DSINQB, VSINQB, VDSINQB`	Sine quarter-wave synthesis
`SINQF, DSINQF, VSINQF, VDSINQF`	Sine quarter-wave transform
`SINQI, DSINQI, VSINQI, VDSINQI`	Initialize sine quarter-wave transform and synthesis

TABLE A-7 FFTPACK and VFFTPACK (Fast Fourier Transform and Vectorized Fast Fourier Transform) Routines *(Continued)*

Routine	Function
`SINT, DSINT, VSINT, VDSINT`	Sine odd-wave transform
`SINTI, DSINT, VSINTI, VDSINTI`	Initialize sine odd-wave transform
`RFFT2B, DFFT2B, CFFT2B, ZFFT2B`	Two-dimensional Fourier synthesis
`RFFT2F, DFFT2F, CFFT2F, ZFFT2F`	Two-dimensional Fourier transform
`RFFT2I, DFFT2I, CFFT2I, ZFFT2I`	Initialize two-dimensional Fourier transform or synthesis
`RFFT3B, DFFT3B, CFFT3B, ZFFT3B`	Three-dimensional Fourier synthesis
`RFFT3F, DFFT3F, CFFT3F, DFFT3F`	Three-dimensional Fourier transform
`RFFT3I, DFFT3I, CFFT3I, ZFFT3I`	Initialize three-dimensional Fourier transform or synthesis
`RFFTOPT,DFFTOPT, CFFTOPT,ZFFTOPT`	Compute the length of the closest FFT

Other Routines

TABLE A-8 Other Routines

Routines	Function
`xCNVCOR`	Computes convolution or correlation
`xCNVCOR2`	Computes two-dimensional convolution or correlation
`xTRANS`	Transposes array
`SWIENER` or `DWEINER`	Performs Wiener deconvolution of two signals

LINPACK Routines

TABLE A-9 LINPACK Routines

Routine	Function
*x*CHDC	Cholesky decomposition of a symmetric positive definite matrix
*x*CHDD	Downdate an augmented Cholesky decomposition
*x*CHEX	Update an augmented Cholesky decomposition with permutations
*x*CHUD	Update an augmented Cholesky decomposition
*x*GBCO	LU Factorization and condition number of a general matrix in banded storage
*x*GBDI	Determinant of an LU-factored general matrix in banded storage
*x*GBFA	LU factorization of a general matrix in banded storage
*x*GBSL	Solution to a linear system in an LU-factored matrix in banded storage
*x*GECO	LU factorization and condition number of a general matrix
*x*GEDI	Determinant and inverse of an LU-factored general matrix
*x*GEFA	LU factorization of a general matrix
*x*GESL	Solution to a linear system in an LU-factored general matrix
*x*GTSL	Solution to a linear system in a tridiagonal matrix
CHICO or ZHICO	UDU factorization and condition number of a Hermitian matrix
CHIDI or ZHIDI	Determinant, inertia, and inverse of a UDU-factored Hermitian matrix
CHIFA or ZHIFA	UDU factorization of a Hermitian matrix
CHISL or ZHISL	Solution to a linear system in a UDU-factored Hermitian matrix
CHPCO or ZHPCO	UDU factorization and condition number of a Hermitian matrix in packed storage
CHPDI or ZHPDI	Determinant, inertia, and inverse of a UDU-factored Hermitian matrix in packed storage
CHPFA or ZHPFA	UDU factorization of a Hermitian matrix in packed storage
CHPSL or ZHPSL	Solution to a linear system in a UDU-factored Hermitian matrix in packed storage
*x*PBCO	Cholesky factorization and condition number of a symmetric positive definite matrix in banded storage

TABLE A-9 LINPACK Routines *(Continued)*

Routine	Function
*x*PBDI	Determinant of a Cholesky-factored symmetric positive definite matrix in banded storage
*x*PBFA	Cholesky factorization of a symmetric positive definite matrix in banded storage
*x*PBSL	Solution to a linear system in a Cholesky-factored symmetric positive definite matrix in banded storage
*x*POCO	Cholesky factorization and condition number of a symmetric positive definite matrix
*x*PODI	Determinant and inverse of a Cholesky-factored symmetric positive definite matrix
*x*POFA	Cholesky factorization of a symmetric positive definite matrix
*x*POSL	Solution to a linear system in a Cholesky-factored symmetric positive definite matrix
*x*PPCO	Cholesky factorization and condition number of a symmetric positive definite matrix in packed storage
*x*PPDI	Determinant and inverse of a Cholesky-factored symmetric positive definite matrix in packed storage
*x*PPFA	Cholesky factorization of a symmetric positive definite matrix in packed storage
*x*PPSL	Solution to a linear system in a Cholesky-factored symmetric positive definite matrix in packed storage
*x*PTSL	Solution to a linear system in a symmetric positive definite tridiagonal matrix
*x*QRDC	QR factorization of a general matrix
*x*QRSL	Solution to a linear system in a QR-factored general matrix
*x*SICO	UDU factorization and condition number of a symmetric matrix
*x*SIDI	Determinant, inertia, and inverse of a UDU-factored symmetric matrix
*x*SIFA	UDU factorization of a symmetric matrix
*x*SISL	Solution to a linear system in a UDU-factored symmetric matrix
*x*SPCO	UDU factorization and condition number of a symmetric matrix in packed storage
*x*SPDI	Determinant, inertia, and inverse of a UDU-factored symmetric matrix in packed storage
*x*SPFA	UDU factorization of a symmetric matrix in packed storage

TABLE A-9 LINPACK Routines *(Continued)*

Routine	Function
x`SPSL`	Solution to a linear system in a UDU-factored symmetric matrix in packed storage
x`SVDC`	Singular value decomposition of a general matrix
x`TRCO`	Condition number of a triangular matrix
x`TRDI`	Determinant and inverse of a triangular matrix
x`TRSL`	Solution to a linear system in a triangular matrix

Index