Contents

Preface 1
Before You Read This Book 1
What Is Not in This Book 1
Related Documents and Web Sites 2
Typographic Conventions 3
Shell Prompts 4
Supported Platforms 4
Accessing Sun Studio Documentation 4
Resources for Developers 6
Contacting Sun Technical Support 6
Sun Welcomes Your Comments 6

1. Introduction 7
   Libraries Included With Sun Performance Library 7
      Netlib 8
   Sun Performance Library Features 9
   Mathematical Routines 10
   Compatibility With Previous LAPACK Versions 10
   Getting Started With Sun Performance Library 11
      Enabling Trap 6 on SPARC Platforms 12
2. **Using Sun Performance Library** 13
   - Improving Application Performance 13
     - Replacing Routines With Sun Performance Library Routines 13
   - Improving Performance of Other Libraries 14
   - Using Tools to Restructure Code 14
   - Fortran Interfaces 14
     - Fortran SUNPERF Module for Use With Fortran 95 15
     - Optional Arguments 17
   - Fortran Examples 18
   - C Interfaces 21
   - C Examples 23

3. **Optimization** 25
   - Using The Sun Performance Library 25
     - Fortran and C 26
     - C++ 26
   - Compiling 26
     - Compiling Code for a 64-Bit Enabled Operating Environments 27
     - 64-Bit Integer Arguments 27

4. **Parallel Processing** 31
   - Run-Time Issues 31
   - Degree of Parallelism 32
   - Synchronization Mechanisms 33
   - Parallel Processing Examples 34

5. **Working With Matrices** 37
   - Matrix Storage Schemes 37
     - Banded Storage 37
     - Packed Storage 38
BLAS1 Routines  151
BLAS2 Routines  152
BLAS3 Routines  153
Sparse BLAS Routines  153
Sparse Solver Routines  155
Signal Processing Library Routines  157
Miscellaneous Signal Processing Routines  160
Interval BLAS (IBLAS) Routines  161
Sort Routines  167

Index  169
# Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 3-1</td>
<td>Comparison of 32-bit and 64-bit Operating Environments</td>
<td>25</td>
</tr>
<tr>
<td>TABLE 6-1</td>
<td>Nettlib Sparse BLAS Naming Conventions</td>
<td>47</td>
</tr>
<tr>
<td>TABLE 6-2</td>
<td>NIST Fortran Sparse BLAS Routine Naming Conventions</td>
<td>48</td>
</tr>
<tr>
<td>TABLE 6-3</td>
<td>SPSOLVE Sparse Solver Routines</td>
<td>49</td>
</tr>
<tr>
<td>TABLE 6-4</td>
<td>SuperLU Computational Routines</td>
<td>63</td>
</tr>
<tr>
<td>TABLE 6-5</td>
<td>SuperLU Utility Routines</td>
<td>63</td>
</tr>
<tr>
<td>TABLE 6-6</td>
<td>Matrix Type Options for DGSSIN() and DGSSFS()</td>
<td>69</td>
</tr>
<tr>
<td>TABLE 6-7</td>
<td>Matrix Ordering Options for DGSSOR() and DGSSFS()</td>
<td>70</td>
</tr>
<tr>
<td>TABLE 7-1</td>
<td>FFT Routines and Their Arguments</td>
<td>78</td>
</tr>
<tr>
<td>TABLE 7-2</td>
<td>Single Precision Linear FFT Routines</td>
<td>81</td>
</tr>
<tr>
<td>TABLE 7-3</td>
<td>Single Precision Two-Dimensional FFT Routines</td>
<td>89</td>
</tr>
<tr>
<td>TABLE 7-4</td>
<td>Single Precision Three-Dimensional FFT Routines</td>
<td>94</td>
</tr>
<tr>
<td>TABLE 7-5</td>
<td>Fast Cosine and Sine Transform Routines and Their Arguments</td>
<td>101</td>
</tr>
<tr>
<td>TABLE 7-6</td>
<td>Convolution and Correlation Routines</td>
<td>114</td>
</tr>
<tr>
<td>TABLE 7-7</td>
<td>Arguments for One-Dimensional Convolution and Correlation Routines SCNVCOR, DCNVCOR, CCNVCOR, and ZCNVCOR</td>
<td>115</td>
</tr>
<tr>
<td>TABLE 7-8</td>
<td>Arguments for Two-Dimensional Convolution and Correlation Routines SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2</td>
<td>116</td>
</tr>
<tr>
<td>TABLE 7-9</td>
<td>Arguments Affecting Minimum Work Array Size for Two-Dimensional Routines: SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2</td>
<td>118</td>
</tr>
<tr>
<td>TABLE 7-10</td>
<td>MYC_INIT and NYC_INIT Dependencies</td>
<td>118</td>
</tr>
</tbody>
</table>
TABLE 7-11 Minimum Dimensions and Data Types for \textit{WORK} Work Array Used With Convolution and Correlation Routines 119

TABLE 8-1 IBLAS Prefixes and Matrix Types 126
TABLE 8-2 Vector Reductions 128
TABLE 8-3 Add or Cancel Vectors 129
TABLE 8-4 Vector Movements 129
TABLE 8-5 Matrix-Vector Operations 129
TABLE 8-6 $O(n^2)$ Matrix Operations 130
TABLE 8-7 $O(n^3)$ Matrix Operations 130
TABLE 8-8 Matrix Movements 130
TABLE 8-9 Vector Set Operations 130
TABLE 8-10 Matrix Set Operations 131
TABLE 8-11 Vector Utilities 131
TABLE 8-12 Matrix Utilities 132
TABLE 8-13 Interval BLAS User-Defined Data Types 132

TABLE A-1 LAPACK (Linear Algebra Package) Routines 136
TABLE A-2 BLAS1 (Basic Linear Algebra Subprograms, Level 1) Routines 151
TABLE A-3 BLAS2 (Basic Linear Algebra Subprograms, Level 2) Routines 152
TABLE A-4 BLAS3 (Basic Linear Algebra Subprograms, Level 3) Routines 153
TABLE A-5 Sparse BLAS Routines 153
TABLE A-6 SPSOLVE Routines 155
TABLE A-7 SuperLU Routines 155
TABLE A-8 FFT Routines 157
TABLE A-9 Sine and Cosine Transform Routines 159
TABLE A-10 Convolution and Correlation Routines 160
TABLE A-11 Convolution and Correlation Routines 160
TABLE A-12 Interval BLAS Routines 161
TABLE A-13 Sort Routines 167
Preface

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™ Studio Fortran 95, C++, and C compilers.

Before You Read This Book

In order to fully use the information in this document, the reader should have a working knowledge of the Fortran or C language and some understanding of the base LAPACK and BLAS libraries available from Netlib (http://www.netlib.org).

What Is Not in This Book

This book does not repeat information included in existing LAPACK books or sources on Netlib. Refer to the next section “Related Documents and Web Sites” on page 2 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 LAPACK and BLAS libraries upon which the Sun Performance Library is based. The *LAPACK Users’ Guide, Third Edition*, Anderson E. and others. SIAM, 1999, augments the material in this manual and provide essential information:


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 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK version 3.1.1</td>
<td><a href="http://www.netlib.org/lapack/">http://www.netlib.org/lapack/</a></td>
</tr>
<tr>
<td>BLAS, levels 1 through 3</td>
<td><a href="http://www.netlib.org/blas/">http://www.netlib.org/blas/</a></td>
</tr>
</tbody>
</table>

*Note* – LINPACK is no longer included in the Sun Performance Library. The LINPACK libraries and documentation are still available from [www.netlib.org](http://www.netlib.org).
# Typographic Conventions

## TYPEFACE CONVENTIONS

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. `% You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td><code>% su Password:</code></td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized</td>
<td>Read Chapter 6 in the <em>User's Guide</em>. These are called <em>class</em> options. You <em>must</em> be superuser to do this.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Command-line placeholder text; replace with a real name or value</td>
<td>To delete a file, type <code>rm filename</code>.</td>
</tr>
</tbody>
</table>

## CODE CONVENTIONS

<table>
<thead>
<tr>
<th>Code Symbol</th>
<th>Meaning</th>
<th>Notation</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ ]</code></td>
<td>Brackets contain arguments that are optional.</td>
<td><code>0[n]</code></td>
<td><code>04, 0</code></td>
</tr>
<tr>
<td><code>{ }</code></td>
<td>Braces contain a set of choices for a required option.</td>
<td>`d(y</td>
<td>n)`</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>The “pipe” or “bar” symbol separates arguments, only one of which may be chosen.</td>
<td>`B{dynamic</td>
</tr>
<tr>
<td><code>:</code></td>
<td>The colon, like the comma, is sometimes used to separate arguments.</td>
<td><code>Rdir[:dir]</code></td>
<td><code>R/local/libs:/U/a</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td>The ellipsis indicates omission in a series.</td>
<td><code>xinline=fl[...ft]</code></td>
<td><code>xinline=alpha,dos</code></td>
</tr>
</tbody>
</table>
Shell Prompts

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell</td>
<td>machine-name%</td>
</tr>
<tr>
<td>C shell superuser</td>
<td>machine-name#</td>
</tr>
<tr>
<td>Bourne shell and Korn shell</td>
<td>$</td>
</tr>
<tr>
<td>Superuser for Bourne shell and Korn shell</td>
<td>#</td>
</tr>
</tbody>
</table>

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at http://www.sun.com/bigadmin/hcl. These documents cite any implementation differences between the platform types.

In this document, these x86 related terms mean the following:

- x86 refers to the larger family of 64-bit and 32-bit x86 compatible products.
- x64 points out specific 64-bit information about AMD64 or EM64T systems.
- 32-bit x86 points out specific 32-bit information about x86 based systems.

For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Documentation

Complete documentation is available from the documentation index at http://developers.sun.com/sunstudio/documentation.
Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

<table>
<thead>
<tr>
<th>Type of Documentation</th>
<th>Format and Location of Accessible Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuals (except third-party manuals)</td>
<td>HTML at <a href="http://docs.sun.com">http://docs.sun.com</a></td>
</tr>
<tr>
<td>Readmes and man pages</td>
<td>HTML through the documentation index at <a href="http://developers.sun.com/sunstudio/documentation">http://developers.sun.com/sunstudio/documentation</a></td>
</tr>
<tr>
<td>Online help</td>
<td>HTML available through the Help menu in the IDE</td>
</tr>
<tr>
<td>Release notes</td>
<td>HTML through the documentation index at <a href="http://developers.sun.com/sunstudio/documentation">http://developers.sun.com/sunstudio/documentation</a></td>
</tr>
</tbody>
</table>

Related Compilers and Tools Documentation

The following table describes related documentation that is available at the http://docs.sun.com website.

<table>
<thead>
<tr>
<th>Document Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Computation Guide</td>
<td>Describes issues regarding the numerical accuracy of floating-point computations.</td>
</tr>
</tbody>
</table>
Resources for Developers

Visit http://developers.sun.com/sunstudio to find these frequently updated resources:
- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of compilers and tools components, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

You can find additional resources for developers at http://developers.sun.com.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:
http://www.sun.com/service/contacting

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL:
http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:
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).

1.1 Libraries Included With Sun Performance Library

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

- LAPACK version 3.1.1 – 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.

Note – LINPACK has been removed from Sun Performance Library. LAPACK version 3.1.1 supersedes LINPACK and all previous versions of LAPACK. If the LINPACK routines are still needed, the LINPACK library and documentation can be obtained from http://www.netlib.org.
Sun Performance Library is available in both static and dynamic library forms. There are optimized SPARC versions for V8, sparcvis, sparcvis2, and sparcfmaf architectures on Solaris 10 operating systems. There are also optimized versions for x86/x64 architectures on Solaris 10 systems, along with SuSE Linux Enterprise Server 9 and Redhat Enterprise Linux 4 systems. All versions have support for parallel programming on multiprocessor platforms. (See the Sun Studio release notes for details.)

Sun Performance Library LAPACK routines have been compiled with a Fortran 95 compiler and remain compatible with the Netlib LAPACK version 3.1.1 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.

1.1.1 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 3.1.1 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 Users’ Guide, Third Edition describes LAPACK algorithms and how to use the routines, but it does not describe the Sun Performance Library extensions made to the base routines.

1.2  Sun Performance Library Features

Sun Performance Library routines can increase application performance on both serial and multiprocessor (MP) platforms, because the serial speed of many Sun Performance Library routines has been increased, and many routines have been parallelized. Sun Performance Library routines also have SPARC, AMD, and Intel 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, LAPACK 2.0, and LAPACK 3.x libraries
- Increased performance, and in some cases, greater accuracy
- Optimizations for specific SPARC and x86/x64 instruction set architectures
- Support for 64-bit enabled Solaris and Linux operating environments
- Support for parallel processing compiler options for SPARC and x86/x64 platforms
- Support for multiple processor hardware options
1.3 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, LDLT and UDUT

- **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

- Interval BLAS routines

- Sorting operations

1.4 Compatibility With Previous LAPACK Versions

The Sun Performance Library routines that are based on LAPACK support the expanded capabilities and improved algorithms in LAPACK 3.1.1, but are completely compatible with both LAPACK 1 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.

### 1.5 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 -xmemalign=8s my_file.c -xlic_lib=sunperf
```

or

```
my_system% CC my_file.cpp -xmemalign=8s -library=sunperf
```

Because Sun Performance Library routines are compiled with -dalign (-xmemalign=8s for C routines), the -dalign option should be used for compilation of all files if any routine in the program makes a Sun Performance Library call. On SPARC platforms, if -dalign cannot be used, enabling Trap 6, described in the section “Enabling Trap 6 on SPARC Platforms” on page 12, is a low-performance workaround that allows misaligned data. While there are no data alignment restrictions on x86/x64 platforms, misaligned data might require extra instructions to properly handle memory transfers, which in turn can cause poor performance.

The -xlic_lib=sunperf option (for F95 and C) and the -library=sunperf option (for C++) include additional compiler and system libraries (e.g. Fortran runtime and micro-tasking library) and set run-time search paths for the resulting executable or shared library.

To summarize, use:
-dalign or -xmemalign=8s on all files at compile time or, on SPARC platforms, enable trap 6
- The same command line options for compiling and linking
- -xlic_lib=sunperf or -library=sunperf

See “Compiling” on page 26, and “Parallel Processing” on page 31 for additional options that optimize application performance.

1.5.1 Enabling Trap 6 on SPARC Platforms

On SPARC platforms, if an application cannot be compiled using -dalign, enable trap 6 to provide a handler for misaligned data. To enable trap 6 on SPARC platforms, 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
```
Using Sun Performance Library

This chapter describes using the Sun Performance Library to improve the execution speed of applications written in 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.

2.1 Improving Application Performance

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

2.1.1 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.
2.1.2 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.

2.1.3 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.

2.2 Fortran Interfaces

Sun Performance Library contains f95 interfaces and legacy f77 interfaces for maintaining compatibility with the standard LAPACK and BLAS libraries and existing codes. Sun Performance Library f95 and legacy f77 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.
2.2.1 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:

```fortran
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:

```fortran
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 Operating Environments” on page 27.

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.

```fortran
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.
```
2.2.2 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 \texttt{SAXPY} routine is defined as follows in the man page.

\begin{verbatim}
SUBROUTINE SAXPY([N], ALPHA, X, [INCX], Y, [INCY])
REAL ALPHA
INTEGER INCX, INCY, N
REAL X(*), Y(*)

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

CALL AXPY(100, ALPHA, X, INCX, Y, INCY)
CALL AXPY(N, RALPHA, XA, INCX, Y, INCY)
\end{verbatim}

The \texttt{N}, \texttt{INCX}, and \texttt{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 \texttt{SAXPY} routine with the following arguments.

\begin{verbatim}
USE SUNPERF
COMPLEX ALPHA
REAL    X(100), Y(100), XA(100,100), RALPHA
INTEGER INCX, INCY

CALL AXPY(100, ALPHA, X, INCX, Y, INCY)
CALL AXPY(N, RALPHA, XA, INCX, Y, INCY)
\end{verbatim}

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

\texttt{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 \texttt{SAXPY} routine due type, shape, or number mismatches.

- \textit{Incorrect type of the arguments}–If \texttt{SAXPY} is called as follows:

\begin{verbatim}
CALL AXPY(100, ALPHA, X, INCX, Y, INCY)
\end{verbatim}

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

- \textit{Incorrect shape of the arguments}– If \texttt{SAXPY} is called as follows:

\begin{verbatim}
CALL AXPY(N, RALPHA, XA, INCX, Y, INCY)
\end{verbatim}

A compiler error occurs because the \texttt{XA} argument is two dimensional, but the interface is expecting a one-dimensional argument.
Incorrect number of arguments—If \texttt{SAXPY} is called as follows:

\begin{verbatim}
CALL AXPY(RALPHA, X, INCX, Y)
\end{verbatim}

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

\begin{verbatim}
AXPY(REAL, REAL 1-D ARRAY, INTEGER, REAL 1-D ARRAY)
\end{verbatim}

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

\begin{verbatim}
CALL AXPY(REAL=RALPHA, X=X, INCX=INCX, Y=Y)
\end{verbatim}

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

The following calls to the \texttt{AXPY} interface are valid.

\begin{verbatim}
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(N, RALPHA, X, Y=Y)
\end{verbatim}

2.3 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 be 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) < 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) < 0.0) THEN
    V2(I,K) = 0.0
  END IF
END DO
CALL DGER (M, N, 1.0D0, X, LDX, Vl(I,K), 1, V2(I,K), 1)
WHERE (V(1:N,K) < 0.0) THEN
  V(1:N,K) = 0.0
END WHERE
CALL DGER (M, N, 1.0D0, X, LDX, Vl(I,K), 1, V2(I,K), 1)
```

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

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 multiprocessor 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
C$PAR DOALL
DO I = 1, N
   CALL DGEMV ('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 DGEMV to multiply a banded matrix by a vector. By putting this routine into a properly constructed loop, 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 multiprocessor platforms can get additional performance by parallelizing these loops.

2.4 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 SPARSE BLAS. 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.

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 one, in conformance with Fortran conventions, rather than being zero as in C.

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 also return a 1, 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.

The SPARC version of the 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.
2.5 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.

```c
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.

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

large_index = isamax (n, a, l) - 1;
largest = a[large_index];
large_index = isamax (n, b, l) - 1;
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 = 1;
/* 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) - 1;
largest = a[large_index]; large_index = isamax_ (&n, b, &one) - 1;
if (b[large_index] > largest)
    largest = b[large_index];
```
This chapter describes how to use compiler and linking options to optimize applications for:

- Specific instruction-set architectures
- 32-bit and 64-bit enabled operating environments

**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>
<thead>
<tr>
<th></th>
<th>32-bit (ILP 32)</th>
<th>64-bit (LP64)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-xarch on SPARC platforms</strong></td>
<td>v8, sparcvis, sparcvis2, sparcfmaf</td>
<td>sparcvis, sparcvis2, sparcfmaf</td>
</tr>
<tr>
<td><strong>-xarch on x86 platforms</strong></td>
<td>generic, sse2</td>
<td>sse2</td>
</tr>
<tr>
<td><strong>addressing</strong></td>
<td>-m32</td>
<td>-m64</td>
</tr>
<tr>
<td><strong>Fortran Integers</strong></td>
<td>INTEGER, INTEGER*4</td>
<td>INTEGER*8</td>
</tr>
<tr>
<td><strong>C Integers</strong></td>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td><strong>Floating-point</strong></td>
<td>S/D/C/Z</td>
<td>S/D/C/Z</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Names of routines</td>
<td>Names of routines with _64 suffix</td>
</tr>
</tbody>
</table>

### 3.1 Using The Sun Performance Library

The Sun Performance Library was compiled using the f95 compiler provided with this release. The Sun Performance Library routines were compiled using `-dalign`, `-xparallel`. 
3.1.1  Fortran and C

When linking the program, use -dalign -xlic_lib=sunperf and the same command line options that were used when compiling.

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
```

3.1.2  C++

When linking your program, use -dalign -library=sunperf, and the same command line options that were used when compiling. Sun Performance Library is linked into an application with the -library switch, as shown here, rather than the -l switch.

```
my_system% CC -dalign my_file.cpp -library=sunperf
```

If -dalign cannot be used in the program, supply a trap 6 handler as described in “Getting Started With Sun Performance Library” on page 11.

3.2  Compiling

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 instruction-set architecture.

**Note** – Using instruction-set specific optimization options improves application performance on the selected instruction set architecture, but limits code portability.

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

The following lists the -xarch values for SPARC instruction-set architectures:

- **SPARC64VI platforms**: Use -xarch=sparcfmaf
UltraSPARC III, IV or IV+ platforms. Use -xarch=sparcvis2.

UltraSPARC I or UltraSPARC II platforms. Use -xarch=sparcvis

The following are the -xarch values for x86 instruction-set architectures:

- AMD Opteron platforms. Use -xarch=sse2
- Intel Core-Duo, AMD Barcelona platforms. Use -xarch=sse3
- Generic x86 systems. Use -xarch=generic

3.2.1 Compiling Code for a 64-Bit Enabled Operating Environments

To compile code for a 64-bit enabled operating environment, use -m64 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 and Fortran 95 code, append _64 to the names of Sun Performance Library routines (for example, rfftf_64 or CFITB_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.

3.2.2 64-Bit Integer Arguments

These additional 64-bit-integer interfaces are available only when linking with -m64. Codes compiled for 32-bit operating environments (-m32) cannot 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.

```c
#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.

```c
#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.

Consider the following code example.

```fortran
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, PARAMETER :: INCX=1, INCY=1, N=10
COMPLEX X(N), Y(N)

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, PARAMETER :: INCX=1, INCY=1, N=10
COMPLEX X(N), Y(N)

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, PARAMETER :: INCX=1, INCY=1, N=10
COMPLEX X(N), Y(N)

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 dcond,
            int *info)
```
The following code example shows calling the `dgbcon` routine using 64-bit arguments.

```c
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

This chapter describes using the Sun Performance Library in multiprocessor environments.

4.1 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 on 32-bit platforms and 8 Mbytes on 64-bit platforms. This is done with the STACKSIZE environment variable (units in Kbytes). To set the per-thread stack size to 4 Mbytes in a 32-bit environment:

```bash
my_host% setenv STACKSIZE 4000
```

To set the per-thread stack size to 8 Mbytes in a 64-bit environment:

```bash
my_host% setenv STACKSIZE 8000
```

Setting the STACKSIZE environment variable is not required for programs running with POSIX or Solaris threads. In this case, user-created threads that call Sun Performance Library routines must have a stack size of at least 4 Mbytes. Failure to supply an adequate stack size for the Sun Performance Library routines might result in stack overflow problems. Symptoms of stack overflow problems include runtime failures that could be difficult to diagnose. For more information on setting the stack size of user-created threads, see the pthread_create(3THR), pthread_attr_init(3THR) and pthread_attr_setstacksize(3THR) man pages for POSIX threads or the thr_create(3THR) for Solaris threads.
4.2 Degree of Parallelism

Selected routines in the Sun Performance Library are parallelized using compiler directives, library routines, and environment variables from the OpenMP Fortran Application Program Interface. The number of threads these routines will perform in parallel is controlled by the environment variable `OMP_NUM_THREADS`, which is set by the user at run time. Environment variable `PARALLEL` can also be used, but if both are set they must have the same value; otherwise, a fatal error will occur upon execution. Both environment variables can be overridden by calling the Sun Performance Library routine `USE_THREADS` or the OpenMP routine `OMP_SET_NUM_THREADS` in the user code.

A user code can be parallelized by:
- setting environment variable `OMP_NUM_THREADS` to a value greater than 1, and
- using compiler parallel directives such as those from the OpenMP API, and/or
- using appropriate compiler flags (`-xopenmp=parallel, -xautopar`).

The Sun Performance Library routines execute in parallel if
- `OMP_NUM_THREADS` is set to a value greater than 1, and
- the routines are not being called from a parallel region

The Sun Performance Library employs OpenMP directives in its parallelization and does not support nested parallelism. If the user code is parallelized as stated above, when a Sun Performance Library routine is called it will execute in serial if it detects that it is being called from a parallel region; otherwise, it will execute in parallel.

POSIX or Solaris threads can also be created to execute in parallel selected regions in the user code. When it is called under this parallel model, a Sun Performance Library routine cannot detect that it is being called from a parallel region. Therefore, the environment variable `OMP_NUM_THREADS` must be set to 1 (or must be unset) or a call to `USE_THREADS()` must be made in appropriate places in the user code. Otherwise, nested parallelism with undefined results will occur.

For example, if the program containing the following code segment is linked with `-xopenmp=parallel` and `OMP_NUM_THREADS` is set to 4, the loop will execute in parallel, and there will be four instances of DGEMM running concurrently. However, each DGEMM instance will run in serial since only one level of parallelization is supported.

```fortran
!$OMP PARALLEL
    DO I = 1, N
        CALL DGEMM(...) 
    END DO 
!$OMP END PARALLEL
```
In the following code example, if the program is not linked with -xautopar, the loop will not be parallelized, but each instance of DGEMM will be executed by four threads.

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

If the program containing the following code segment is linked with -xopenmp=parallel and if OMP_NUM_THREADS is set to a value greater than 1, the region shown will be executed by a single thread. However, each DGEMM call will be executed by OMP_NUM_THREADS threads.

```fortran
!$OMP SINGLE
    DO I = 1, N
        CALL DGEMM(...)
    END DO
!$OMP END SINGLE
```

In the following code example, there will be at most two-way parallelism, regardless of the number of OpenMP threads available for execution. Only one level of parallelism exists, which are the two sections. Further parallelism within a DGEMM call is suppressed.

```fortran
!$OMP PARALLEL SECTIONS
    !$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
```

### 4.3 Synchronization Mechanisms

One characteristic of the POSIX/Solaris threading model is that bound threads of a running application relinquish the CPUs when they are idle, thus providing good throughput and resource usage in a shared (over-subscribed) environment. By
default, bound threads in a compiler-parallelized code spin-wait when they are idle, which can result in suboptimal throughput when there are other applications in the system competing for CPU resource. In this case, environment variable SUNW_MP_THR_IDLE can be used to control the behavior of a thread after it finishes its share of a parallel job:

```
my_host% setenv SUNW_MP_THR_IDLE value
```

Here, value can either be spin or sleep n s or sleep n ms, and spin is the default. sleep puts the thread to sleep after spin-waiting n units. The wait unit can be seconds (s, the default unit) or milliseconds (ms). sleep with no arguments puts the thread to sleep immediately after completing a parallel task. If SUNW_MP_THR_IDLE contains an illegal value or isn’t set, spin is used as the default.

The following settings would cause threads to spin-wait (default behavior), spin for 2 seconds before sleeping, or spin for 100 milliseconds before sleeping, respectively. Using Sun Performance Library routines does not change the spin-wait behavior of the code.

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

4.4 Parallel Processing Examples

This section demonstrates using the OMP_NUM_THREADS environment variable along with compile and link options to create code that execute serially and in parallel.

To create a serial application:

- Call one or more Sun Performance Library routines
- Link with -xlic_lib=sunperf, placing the flag at the end of the command line. Do not compile or link with -xopenmp=parallel, or -xautopar
- Unset OMP_NUM_THREADS environment variable or set it to 1

The following example shows how to compile and link with the shared Sun Performance library libsunperf.so.

```
my_host% cc -xmemalign=8s -xarch=native any.c -xlic_lib=sunperf
```
or

```
my_host% f95 -dalign -xarch=native any.f95 -xlic_lib=sunperf
```

To create a parallel application that execute on multiple processors:

- Call one or more Sun Performance Library routines
- Use the same parallelization option (-xopenmp=parallel or -xautopar) in the compile and link commands
- Link with -xlic_lib=sunperf, placing the flag at the end of the command line
- Set OMP_NUM_THREADS to the number of available processors before running the executable

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

```
my_host% f95 -dalign -xarch=native my_app.f -xlic_lib=sunperf
my_host% setenv OMP_NUM_THREADS 24
my_host% ./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 -xopenmp=parallel or -xautopar is required on the compile line:

```
my_host% f95 -dalign -xopenmp=parallel my_app.f -xlic_lib=sunperf
my_host% setenv OMP_NUM_THREADS 24
my_host% ./a.out
```
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.

### 5.1 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.

#### 5.1.1 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
C Copy the matrix A from the array AG to the array AB. The matrix is stored in general storage mode in AG and it will be stored in banded storage mode in AB. The code to copy from general to banded storage mode is taken from the 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
```

This method of storing banded matrices is compatible with the storage method used by LAPACK and BLAS.

### 5.1.2 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 $A_P$ as shown below. This code comes from the comment block of the LAPACK routine DTPTRI.

```c
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
```

5.2 Matrix Types

The general matrix is the most common type, and most operations in the Sun Performance Library operate on the general matrix. In many cases, there are routines that will work with the other types of matrices. For example, DGEMM computes the product of two general matrices, and DTRMM computes the product of a triangular matrix and a general matrix.

5.2.1 General Matrices

The storage of a general matrix is such that there is a one-to-one correspondence between the elements of the matrix and the elements of the array. Element Aij of matrix A is stored in element A(I,J) of the corresponding array A. The general matrix has no special storage scheme since each of its elements is stored explicitly. In contrast, only the nonzero upper-diagonal, diagonal, and lower-diagonal elements of
a general band matrix are stored. The following example shows how a general band matrix is stored in a two-dimensional array. Array locations marked with x are not accessed.

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

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

---

**5.2.2 Triangular Matrices**

There are two storage schemes for a triangular matrix. In the unpacked scheme where the matrix is stored in a two-dimensional array, there is a one-to-one correspondence between all elements of the matrix and the elements of the array, but zero entries in the matrix are neither set nor accessed in the array (denoted by x). In the packed storage scheme, nonzero elements of the matrix are packed by column in a one-dimensional array.

A triangular matrix can be stored using packed storage.

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

\[
\begin{bmatrix}
a_{11} & x & x \\
a_{21} & a_{22} & x \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

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

---

General Band Matrix  | General Band Matrix in Packed Storage
---|---

Triangular Band Matrix  | Triangular Matrix in Unpacked Storage  | Triangular Matrix in Packed Storage
A triangular band matrix can be stored in packed storage using a two-dimensional array as shown below. Elements marked with x are not accessed.

\[
\begin{bmatrix}
  a_{11} & 0 & 0 \\
  a_{21} & a_{22} & 0 \\
  0 & a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
  a_{11} & a_{22} & a_{33} \\
  a_{21} & a_{32} & x
\end{bmatrix}
\]

#### 5.2.3 Symmetric Matrices

A real symmetric or complex Hermitian matrix is similar to a triangular matrix in that only elements in its upper or lower triangle is explicitly stored in the corresponding elements of a two-dimensional array. The remaining elements of the array (denoted by x below) are neither set nor accessed. The active upper or lower triangle can also be packed by column into a one-dimensional array.

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

- **Symmetric Matrix**
- **Symmetric Matrix in Unpacked Storage**
- **Symmetric Matrix in Packed Storage**
## 5.2.4 Tridiagonal Matrices

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

\[
\begin{bmatrix}
a_{11} & a_{12} & 0 & 0 \\
a_{21} & a_{22} & a_{23} & 0 \\
0 & a_{32} & a_{33} & a_{34} \\
0 & 0 & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
a_{21} \\
a_{32} \\
a_{43}
\end{bmatrix}
\begin{bmatrix}
a_{11} \\
a_{22} \\
a_{33} \\
a_{44}
\end{bmatrix}
\]

| Tridiagonal Matrix | Storage for Tridiagonal Matrix |
The Sun Performance Library has two software packages, SPSOLVE and SuperLU, that can be used to factor and solve sparse linear systems of equations.

Mainly written in Fortran, SPSOLVE is a collection of routines that solve symmetric, structurally symmetric, and unsymmetric coefficient matrices, using one of several ordering methods, including a user-specified ordering. In previous releases, SPSOLVE was referred to as the sparse solver package. It contains interfaces for FORTRAN 77; Fortran 95 and C interfaces are not currently provided. To use SPSOLVE routines from Fortran 95, use the FORTRAN 77 interfaces. To call SPSOLVE from C, append an underscore to the routine name (dgssin_(), dgssor_(), and so on), pass arguments by reference, and use one-based array indexing. (See “Unsymmetric Sparse Matrices” on page 45 for an example of one-based and zero-based array indexing. For information on how to call Fortran routines from C, see the Fortran Programming Guide.)

The SuperLU package in the Sun Performance Library is the sequential version (version 3.0) of the public domain application that solves general unsymmetric sparse systems. While it is sequential, SuperLU does make use of several level 2 and level 3 BLAS routines that are parallelized. For detailed documentation of its algorithm, routines and data structures, see [5, 6, 7]. SuperLU is written in C; therefore, array indexing must be zero-based regardless of whether its routines are being called from Fortran-based SPSOLVE or a C driver program. (See SuperLU Interface for more detail and examples.)
6.1 Sparse Matrices

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 SPSOLVE and SuperLU 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

Currently, SuperLU only supports unsymmetric matrices. The most efficient data representation often depends on the specific problem. The following sections show examples of sparse matrix data formats.

6.1.1 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.”
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

6.1.2 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.

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

To represent $A$ in CSC format:
- colptr: 1, 6, 7, 8, 9
- rowind: 1, 2, 3, 4, 5
- values: 1.0, 3.0, 0.0, 0.0

6.1.3 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.

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

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
To represent $A$ in CSC format:

- **One-based indexing:**
  - 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

- **Zero-based indexing:**
  - colptr: 0, 5, 6, 7, 8, 10
  - rowind: 0, 1, 2, 3, 4, 1, 2, 3, 1, 4
  - values: 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0

6.2 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 NIST Fortran Sparse BLAS routines, refer to [http://math.nist.gov/spblas/](http://math.nist.gov/spblas/)

The Netlib Sparse BLAS and NIST Fortran Sparse BLAS Library routines each use their own naming conventions, as described in the following sections.
6.2.1 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 6-1 lists the naming conventions for the Netlib Sparse BLAS vector routines.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Root of Name</th>
<th>Prefix and Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot product</td>
<td>-DOT-</td>
<td>S-I D-I C-UI Z-UI C-CI Z-CI</td>
</tr>
<tr>
<td>Scalar times a vector added to a vector</td>
<td>-AXPY-</td>
<td>S-I D-I C-I Z-I</td>
</tr>
<tr>
<td>Apply Givens rotation</td>
<td>-ROT-</td>
<td>S-I D-I</td>
</tr>
<tr>
<td>Gather x into y</td>
<td>-GTHR-</td>
<td>S- D- C- Z- S-Z D-Z C-Z Z-Z</td>
</tr>
<tr>
<td>Scatter x into y</td>
<td>-SCTR-</td>
<td>S- D- C- Z-</td>
</tr>
</tbody>
</table>

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.

6.2.2 NIST Fortran Sparse BLAS

Each NIST Fortran Sparse BLAS routine has a six-character name of the form XYYYYZZ where:
- X represents the data type.
- YYY represents the sparse storage format.
- ZZ represents the operation.
TABLE 6-2 shows the values for X, Y, and Z.

<table>
<thead>
<tr>
<th>X: Data Type</th>
<th>S: single precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>D: double precision</td>
<td></td>
</tr>
<tr>
<td>C: complex</td>
<td></td>
</tr>
<tr>
<td>Z: double complex</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YYY: Sparse Storage Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single entry formats:</td>
</tr>
<tr>
<td>COO: coordinate</td>
</tr>
<tr>
<td>CSC: compressed sparse column</td>
</tr>
<tr>
<td>CSR: compressed sparse row</td>
</tr>
<tr>
<td>DIA: diagonal</td>
</tr>
<tr>
<td>ELL: ellpack</td>
</tr>
<tr>
<td>JAD: jagged diagonal</td>
</tr>
<tr>
<td>SKY: skyline</td>
</tr>
<tr>
<td>Block entry formats:</td>
</tr>
<tr>
<td>BCO: block coordinate</td>
</tr>
<tr>
<td>BSC: block compressed sparse column</td>
</tr>
<tr>
<td>BSR: block compressed sparse row</td>
</tr>
<tr>
<td>BDI: block diagonal</td>
</tr>
<tr>
<td>BEL: block ellpack</td>
</tr>
<tr>
<td>VBR: block compressed sparse row</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ZZ: Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM: matrix-matrix product</td>
</tr>
<tr>
<td>SM: solution of triangular system (supported for all formats except COO)</td>
</tr>
<tr>
<td>RP: right permutation (for JAD format only)</td>
</tr>
</tbody>
</table>

### 6.3 SPSOLVE Interface

SPSOLVE computes the solution of a sparse system through a sequence of steps: Initialization, ordering to reduce fill-in, symbolic factorization, numeric factorization, and triangular solve. A user code may call individual routines or make use of a one-call interface to perform these steps.
6.3.1 SPSOLVE Routines

Listed in the table below are user-accessible routines in SPSOLVE and their functions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGSSFS()</td>
<td>One call interface to sparse solver</td>
</tr>
<tr>
<td>DGSSIN()</td>
<td>Sparse solver initialization</td>
</tr>
<tr>
<td>DGSSOR()</td>
<td>Fill reducing ordering and symbolic factorization</td>
</tr>
<tr>
<td>DGSSSUO()</td>
<td>Sets user-specified ordering permutation and performs symbolic</td>
</tr>
<tr>
<td></td>
<td>factorization (called in place of DGSSOR)</td>
</tr>
<tr>
<td>DGSSFA()</td>
<td>Matrix value input and numeric factorization</td>
</tr>
<tr>
<td>DGSSSL()</td>
<td>Triangular solve</td>
</tr>
<tr>
<td>DGSSRP()</td>
<td>Returns permutation used by solver.</td>
</tr>
<tr>
<td>DGSSCO()</td>
<td>Returns condition number estimate of coefficient matrix.</td>
</tr>
<tr>
<td>DGSSDA()</td>
<td>De-allocates sparse solver.</td>
</tr>
<tr>
<td>DGSSPS()</td>
<td>Prints solver statistics.</td>
</tr>
</tbody>
</table>

Matrices with the same structure but with different numerical values can be solved by calling SPSOLVE routines in the order shown below:

```
call dgssin() ! initialization, input coefficient matrix structure
call dgssor() ! fill-reducing ordering, symbolic factorization
   ! (or call dgssuo() to specify a user ordering,
   ! and perform 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
endo
doendo
```
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 \texttt{dgsssl()} are used to solve for additional right-hand sides, as shown in the following example.

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

### 6.3.2 Routine Calling Order

To use SPSOLVE, its routines must be called in the order shown below:

1. One Call Interface: For solving single matrix
   a. \texttt{DGSSFS()} - Initialize, order, factor, solve
   b. \texttt{DGSSSL()} - Additional solves (optional): repeat \texttt{DGSSSL()} as needed
   c. \texttt{DGSSDA()} - Deallocate working storage

2. Regular Interface: For solving multiple matrices with the same structure
   a. \texttt{DGSSIN()} - Initialize
   b. \texttt{DGSSOR()} or \texttt{DGSSUO()} - Order and symbolically factor
   c. \texttt{DGSSFA()} - Factor
   d. \texttt{DGSSSL()} - Solve: repeat \texttt{DGSSFA()} or \texttt{DGSSSL()} as needed
   e. \texttt{DGSSDA()} - Deallocate working storage

### 6.3.3 SPSOLVE Examples

The following examples show solving a symmetric system using the one-call interface, and solving a symmetric system using the regular interface. In \textit{CODE EXAMPLE 6-1}, the one-call interface is used to solve a symmetric system, and in \textit{CODE EXAMPLE 6-2}, individual routines are called to solve a symmetric system.
CODE EXAMPLE 6-5 shows how the Fortran SPSOLVE interface can be called from a C program. (For more information on how to call Fortran routines from C programs, see the Fortran Programming Guide.)

CODE EXAMPLE 6-1 Solving a Symmetric System—One-Call Interface

```c
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.
    c
    A x = b, (solve for x) where:
    c
    4.0  1.0  2.0  0.5  2.0  2.0  7.0
    1.0  0.5  0.0  0.0  0.0  2.0  3.0
    A = 2.0  0.0  3.0  0.0  0.0  x = 1.0  b = 7.0
    0.5  0.0  0.0  0.625 0.0  -8.0  -4.0
    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, 0.625d0, &
                  0.0d0, 3.0d0, 0.0d0, 0.0d0, 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'
```

Chapter 6 Sparse Computation 6-51
CODE EXAMPLE 6-1  Solving a Symmetric System–One-Call Interface (Continued)

```c
 neqns  = 5
 nrhs   = 1

 ldrhs  = 5
 outunt = 6
 msglvl1 = 0
 ordmthd = 'mmd'

 call single call interface

 call dgssfs ( mtxtyp, pivot, neqns, colstr, rowind,
 &              values, nrhs, rhs, ldrhs, ordmthd,
 &              outunt, msglvl1, handle, ier             )
 if ( ier .ne. 0 ) goto 110

 deallocate sparse solver storage

 call dgssda ( handle, ier )
 if ( ier .ne. 0 ) goto 110

 print values of sol

 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

 call to sparse solver returns an error

 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
```
**CODE EXAMPLE 6-1  Solving a Symmetric System–One-Call Interface (Continued)**

```plaintext
my_system% f95 -dalign example_lcall.f -xlic_lib=sunperf
my_system% a.out

<table>
<thead>
<tr>
<th>i</th>
<th>rhs(i)</th>
<th>expected rhs(i)</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.200000000000D+01</td>
<td>0.200000000000D+01</td>
<td>-0.528466159722D-13</td>
</tr>
<tr>
<td>2</td>
<td>0.200000000000D+01</td>
<td>0.200000000000D+01</td>
<td>0.105249142734D-12</td>
</tr>
<tr>
<td>3</td>
<td>0.100000000000D+01</td>
<td>0.100000000000D+01</td>
<td>0.350830475782D-13</td>
</tr>
<tr>
<td>4</td>
<td>-0.800000000000D+01</td>
<td>-0.800000000000D+01</td>
<td>0.426325641456D-13</td>
</tr>
<tr>
<td>5</td>
<td>-0.500000000000D+00</td>
<td>-0.500000000000D+00</td>
<td>0.660582699652D-14</td>
</tr>
</tbody>
</table>
```

**CODE EXAMPLE 6-2  Solving a Symmetric System–Regular Interface**

```plaintext
my_system% cat example_ss.f
program example_ss

 This program is an example driver that calls the sparse solver.
 It factors and solves a symmetric system.

 implicit none

 integer negns, 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

 Sparse matrix structure and value arrays. From George and Liu,
 page 3.
 Ax = b, (solve for x) where:

 A = 4.0 1.0 2.0 0.5 2.0 2.0
 1.0 0.5 0.0 0.0 0.0 2.0
 2.0 0.0 3.0 0.0 0.0 x = 1.0
 0.5 0.0 0.0 0.625 0.0
 2.0 0.0 0.0 0.0 16.0

 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 /
```
CODE EXAMPLE 6-2  Solving a Symmetric System–Regular Interface (Continued)

```c
    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  initialize solver
    c
    mtxtyp= 'ss'
    pivot = 'n'
    neqns  = 5
    outunt = 6
    msglvl = 0
    c  call regular interface
    c
    call dgssin ( mtxtyp, pivot,  neqns , colstr, rowind,  
                & outunt, msglvl, handle, ier    )
    if ( ier .ne. 0 ) goto 110
    c  ordering and symbolic factorization
    c
    ordmthd = 'mmd'
    call dgssor ( ordmthd, handle, ier )
    if ( ier .ne. 0 ) goto 110
    c  numeric factorization
    c
    call dgssfa ( neqns, colstr, rowind, values, handle, ier )
    if ( ier .ne. 0 ) goto 110
    c  solution
    c
    nrhs   = 1
    ldrhs  = 5
    call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
    if ( ier .ne. 0 ) goto 110
    c  deallocate sparse solver storage
    c
    call dgssda ( handle, ier )
    if ( ier .ne. 0 ) goto 110
    c  print values of sol
```
Chapter 6 Sparse Computation

CODE EXAMPLE 6-2  Solving a Symmetric System—Regular Interface (Continued)

```fortran
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
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_system% a.out

<table>
<thead>
<tr>
<th>i</th>
<th>rhs(i)</th>
<th>expected rhs(i)</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.200000000000D+01</td>
<td>0.200000000000D+01</td>
<td>-0.528466159722D-13</td>
</tr>
<tr>
<td>2</td>
<td>0.200000000000D+01</td>
<td>0.200000000000D+01</td>
<td>0.105249142734D-12</td>
</tr>
<tr>
<td>3</td>
<td>0.100000000000D+01</td>
<td>0.100000000000D+01</td>
<td>0.350830475782D-13</td>
</tr>
<tr>
<td>4</td>
<td>-0.800000000000D+01</td>
<td>-0.800000000000D+01</td>
<td>0.426325641456D-13</td>
</tr>
<tr>
<td>5</td>
<td>-0.500000000000D+00</td>
<td>-0.500000000000D+00</td>
<td>0.660582699652D-14</td>
</tr>
</tbody>
</table>

CODE EXAMPLE 6-3  Solving a Structurally Symmetric System With Unsymmetric Values—Regular Interface

```fortran
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).
```
CODE EXAMPLE 6-3  Solving a Structurally Symmetric System With Unsymmetric Values—Regular Interface (Continued)

```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  Sparse matrix structure and value arrays. Coefficient matrix
  has a symmetric structure and unsymmetric values.
  \( Ax = b \), (solve for x) where:

c  \[
  \begin{pmatrix}
  1.0 & 3.0 & 0.0 & 0.0 & 1.0 & 7.0 \\
  2.0 & 4.0 & 0.0 & 7.0 & 2.0 & 38.0 \\
  0.0 & 0.0 & 6.0 & 0.0 & x = 3.0 & b = 18.0 \\
  0.0 & 5.0 & 0.0 & 8.0 & 4.0 & 42.0
  \end{pmatrix}
  \]

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  initialize solver

c      mtxtyp= 'su'
pivot = 'n'
neqns  = 4
outunt = 6
msglvl = 0

c  call regular interface

c      call dgssin ( mtxtyp, pivot, neqns , colstr, rowind, 
&          outunt, msglvl, handle, ier )
if ( ier .ne. 0 ) goto 110
```

ordering and symbolic factorization

ordmthd = 'mmd'
call dgssor ( ordmthd, handle, ier )
if ( ier .ne. 0 ) goto 110

numeric factorization

call dgssfa ( neqns, colstr, rowind, values, handle, ier )
if ( ier .ne. 0 ) goto 110

solution

nrhs   = 1
ldrhs  = 4
call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
if ( ier .ne. 0 ) goto 110

deallocate sparse solver storage

call dgssda ( handle, ier )
if ( ier .ne. 0 ) goto 110

print values of sol

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))
endo
stop

110 continue

call to sparse solver returns an error

cwrite ( 6 , 400 )
& ' example: FAILED sparse solver error number = ', ier
stop

200 format(a5,3a20)
CODE EXAMPLE 6-3  Solving a Structurally Symmetric System With Unsymmetric Values–Regular Interface (Continued)

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

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

end
```

```fortran
my_system% f95 -dalign example_su.f -xlic_lib=sunperf
```

```fortran
my_system% a.out
```

<table>
<thead>
<tr>
<th>i</th>
<th>rhs(i) expected rhs(i) error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.100000000000D+01 0.100000000000D+01 0.000000000000D+00</td>
</tr>
<tr>
<td>2</td>
<td>0.200000000000D+01 0.200000000000D+01 0.000000000000D+00</td>
</tr>
<tr>
<td>3</td>
<td>0.300000000000D+01 0.300000000000D+01 0.000000000000D+00</td>
</tr>
<tr>
<td>4</td>
<td>0.400000000000D+01 0.400000000000D+01 0.000000000000D+00</td>
</tr>
</tbody>
</table>

CODE EXAMPLE 6-4  Solving an Unsymmetric System–Regular Interface

```fortran
my_system% cat example_uu.f
```

```fortran
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. Unsymmetric 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
```
CODE EXAMPLE 6-4  Solving an Unsymmetric System–Regular Interface (Continued)

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 /

initialize solver

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

ordering and symbolic factorization

ordmthd = 'mmd'
call dgssor ( ordmthd, handle, ier )
if ( ier .ne. 0 ) goto 110

numeric factorization

call dgssfa ( neqns, colstr, rowind, values, handle, ier )
if ( ier .ne. 0 ) goto 110

solution

nrhs = 1
ldrhs = 5
call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
if ( ier .ne. 0 ) goto 110

deallocate sparse solver storage

call dgssda ( handle, ier )
if ( ier .ne. 0 ) goto 110
CODE EXAMPLE 6-4  Solving an Unsymmetric System—Regular Interface (Continued)

```c
print values of sol
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

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

```c
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
```

<table>
<thead>
<tr>
<th>i</th>
<th>rhs(i)</th>
<th>expected rhs(i)</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.100000000000D+01</td>
<td>0.100000000000D+01</td>
<td>0.000000000000D+00</td>
</tr>
<tr>
<td>2</td>
<td>0.200000000000D+01</td>
<td>0.200000000000D+01</td>
<td>0.000000000000D+00</td>
</tr>
<tr>
<td>3</td>
<td>0.300000000000D+01</td>
<td>0.300000000000D+01</td>
<td>0.000000000000D+00</td>
</tr>
<tr>
<td>4</td>
<td>0.400000000000D+01</td>
<td>0.400000000000D+01</td>
<td>0.000000000000D+00</td>
</tr>
<tr>
<td>5</td>
<td>0.500000000000D+01</td>
<td>0.500000000000D+01</td>
<td>0.000000000000D+00</td>
</tr>
</tbody>
</table>

CODE EXAMPLE 6-5  Calling SPSOLVE Routines From C

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sunperf.h>

int main() {
    /*
    Sparse matrix structure and value arrays. Coefficient matrix
    */
```
is a general unsymmetric sparse matrix.

\[ Ax = b, \text{ (solve for } x) \text{ where:} \]

\[
\begin{align*}
1.0 & \quad 0.0 & \quad 7.0 & \quad 9.0 & \quad 0.0 & \quad 1.0 & \quad 17.0 \\
2.0 & \quad 4.0 & \quad 0.0 & \quad 0.0 & \quad 0.0 & \quad 1.0 & \quad 6.0 \\
A = 0.0 & \quad 5.0 & \quad 8.0 & \quad 0.0 & \quad 0.0 & \quad x = 1.0 & \quad b = 13.0 \\
0.0 & \quad 0.0 & \quad 0.0 & \quad 10.0 & \quad 11.0 & \quad 1.0 & \quad 21.0 \\
3.0 & \quad 6.0 & \quad 0.0 & \quad 0.0 & \quad 12.0 & \quad 1.0 & \quad 21.0
\end{align*}
\]

/*
/* Array indices must be one-based for calling SPSOLVE routines */
int colstr[] = {1, 4, 7, 9, 11, 13};
int rowind[] = {1, 2, 5, 2, 3, 5, 1, 3, 1, 4, 4, 5};
double values[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0};
double rhs[] = {17.0, 6.0, 13.0, 21.0, 21.0};
double xexpct[] = {1.0, 1.0, 1.0, 1.0, 1.0};
int n = 5, nnz = 12, nrhs = 1, msglvl = 0, outunt = 6, ierr,
    i,j,k, int_ierr;
double t[4], handle[150];
char type[] = "uu", piv = 'n';

/* Last two parameters in argument list indicate lengths of
* character arguments type and piv */
dgssin_(type, &piv, &n, colstr, rowind, &outunt, &msglvl,
    handle, &ierr,2,1);
if (ierr != 0) {
    int_ierr = ierr;
    printf("dgssin err = %d\n", int_ierr);
    return -1;
}
char ordmth[] = "mmd";
dgssor_(ordmth, handle, &ierr, 3);
if (ierr != 0) {
    int_ierr = ierr;
    printf("dgssor err = %d\n", int_ierr);
    return -1;
}
dgssfa_(&n, colstr, rowind, values, handle, &ierr);
6.4 SuperLU Interface

SuperLU has two driver routines, simple and expert, that can be called to completely solve a general unsymmetric sparse system in a similar manner to the one-call interface in SPSOLVE. These and other SuperLU user-callable routines are available in single precision, double precision, complex and double complex data types. Single precision names of all external routines are listed in the following tables. Man...
pages (section 3P) are available for these routines. Also see the man page of SuperMatrix for a description of the sparse matrix data structure that is used in the application.

**TABLE 6-4**  SuperLU Computational Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sgstrf</td>
<td>Computes factorization</td>
</tr>
<tr>
<td>sgssvx</td>
<td>Factorizes and solves (expert driver)</td>
</tr>
<tr>
<td>sgssv</td>
<td>Factorizes and solves (simple driver)</td>
</tr>
<tr>
<td>sgstrs</td>
<td>Computes triangular solve</td>
</tr>
<tr>
<td>sgtrfs</td>
<td>Improves computed solution; provides error bounds</td>
</tr>
<tr>
<td>slangs</td>
<td>Computes one-norm, Frobenius-norm, or infinity-norm</td>
</tr>
<tr>
<td>sgsequ</td>
<td>Computes row and column scalings</td>
</tr>
<tr>
<td>sgsequ</td>
<td>Estimates reciprocal of condition number</td>
</tr>
<tr>
<td>slaqgs</td>
<td>Equilibrates a general sparse matrix</td>
</tr>
</tbody>
</table>

**TABLE 6-5**  SuperLU Utility Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUSolveTime</td>
<td>Returns time spent in solve stage</td>
</tr>
<tr>
<td>LUFactTime</td>
<td>Returns time spent in factorization stage</td>
</tr>
<tr>
<td>LUFactFlops</td>
<td>Returns number of floating point operations in factorization stage</td>
</tr>
<tr>
<td>LUSolveFlops</td>
<td>Returns number of floating point operations in solve stage</td>
</tr>
<tr>
<td>sQuerySpace</td>
<td>Returns information on the memory statistics</td>
</tr>
<tr>
<td>sp_ienv</td>
<td>Returns specified machine dependent parameter</td>
</tr>
<tr>
<td>sPrintPerf</td>
<td>Prints statistics collected by the computational routines</td>
</tr>
<tr>
<td>set_default_options</td>
<td>Sets parameters that control solver behavior to default options</td>
</tr>
<tr>
<td>StatInit</td>
<td>Allocates and initializes structure that stores performance statistics</td>
</tr>
<tr>
<td>StatFree</td>
<td>Frees structure that stores performance statistics</td>
</tr>
<tr>
<td>Destroy_Dense_Matrix</td>
<td>Deallocates a SuperMatrix in dense format</td>
</tr>
<tr>
<td>Destroy_SuperNode_Matrix</td>
<td>Deallocates a SuperMatrix in supernodal format</td>
</tr>
</tbody>
</table>
### TABLE 6-5  SuperLU Utility Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destroy_CompCol_Matrix</td>
<td>Deallocates a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>Destroy_CompCol_Permuted</td>
<td>Deallocates a SuperMatrix in permuted compressed sparse column format</td>
</tr>
<tr>
<td>Destroy_SuperMatrix_Store</td>
<td>Deallocates actual storage that stores matrix in a SuperMatrix</td>
</tr>
<tr>
<td>sCopy_CompCol_Matrix</td>
<td>Copies a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>sCreate_CompCol_Matrix</td>
<td>Allocates a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>sCreate_Dense_Matrix</td>
<td>Allocates a SuperMatrix in dense format</td>
</tr>
<tr>
<td>sCreate_CompRow_Matrix</td>
<td>Allocates a SuperMatrix in compressed sparse row format</td>
</tr>
<tr>
<td>sCreate_SuperNode_Matrix</td>
<td>Allocates a SuperMatrix in supernodal format</td>
</tr>
<tr>
<td>sp_preorder</td>
<td>Permutes columns of original sparse matrix</td>
</tr>
<tr>
<td>sp_sgemm</td>
<td>Multiplies a SuperMatrix by a dense matrix</td>
</tr>
</tbody>
</table>
6.4.1 Calling from C

SuperLU routines are written in C. Therefore, column- and row-related indices must be zero-based. In the following example, double precision simple driver dgssv is called to compute factors L and U and to solve for the solution matrix.

CODE EXAMPLE 6-6 SuperLU Simple Driver

```c
#include <stdio.h>
#include <sunperf.h>
#define M 5
#define N 5

int main(int argc, char *argv[]) {
    SuperMatrix A, L, U, B1, B2;
    int perm_r[M]; /* row permutations from partial pivoting */
    int perm_c[N]; /* column permutation vector */
    int info, i;
    superlu_options_t options;
    SuperLUStat_t stat;
    trans_t trans = NOTRANS;

    printf("Example code calling SuperLU simple driver to factor a \
            general unsymmetric matrix and solve two right-hand-side matrices\n");

    /* the matrix in Harwell-Boeing format. */
    int m = M;
    int n = M;
    int nnz = 12;
    double *dp;
    /* nonzeros of A, column-wise */
    double a[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
                  7.0, 8.0, 9.0, 10.0, 11.0, 12.0};
    /* row index of nonzeros */
    int asub[] = {0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4};
    /* column pointers */
    int xa[] = {0, 3, 6, 8, 10, 12};

    /* Create Matrix A in the format expected by SuperLU */
    dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, SLU_NC, SLU_D, SLU_GE);

    int nrhs = 1;
    double rhs1[] = {17.0, 6.0, 13.0, 21.0, 21.0};
```


```c
/* right-hand side matrix B1, B2 */
dCreate_Dense_Matrix(&B1, m, nrhs, rhs1, m, SLU_DN, SLU_D, SLU_GE);
dCreate_Dense_Matrix(&B2, m, nrhs, rhs2, m, SLU_DN, SLU_D, SLU_GE);

/* set options that control behavior of solver to default parameters */
set_default_options(&options);
options.ColPerm = NATURAL;

/* Initialize the statistics variables. */
StatInit(&stat);
/* factor input matrix and solve the first right-hand-side matrix */
dgssv(&options, &A, perm_c, perm_r, &L, &U, &B1, &stat, &info);

printf("\nsolution matrix B1:");
dp = (double *) (((NCformat *)B1.Store)->nzval);
printf("    i    rhs[i]     expected\n");
for (i=0; i<M; i++)
    printf("%5d   %7.4lf     %7.4lf\n", i, dp[i], 1.0);
printf("Factor time = %8.2e sec\n", stat.utime[FACT]);
printf("Solve time   = %8.2e sec\n\n", stat.utime[SOLVE]);

/* solve the second right-hand-side matrix */
dgstrs(trans, &L, &U, perm_c, perm_r, &B2, &stat, &info);

printf("solution matrix B2:\n");
dp = (double *) (((NCformat *)B2.Store)->nzval);
printf("    i    rhs[i]     expected\n");
for (i=0; i<M; i++)
    printf("%5d   %7.4lf     %7.4lf\n", i, dp[i], 0.3);
printf("Solve time   = %8.2e sec\n", stat.utime[SOLVE]);
StatFree(&stat);
Destroy_CompCol_Matrix(&A);
Destroy_SuperMatrix_Store(&B1);
Destroy_SuperMatrix_Store(&B2);
Destroy_SuperNode_Matrix(&L);
Destroy_CompCol_Matrix(&U);
```

**CODE EXAMPLE 6-6**  SuperLU Simple Driver  *(Continued)*

/* right-hand side matrix B1, B2 */
dCreate_Dense_Matrix(&B1, m, nrhs, rhs1, m, SLU_DN, SLU_D, SLU_GE);
dCreate_Dense_Matrix(&B2, m, nrhs, rhs2, m, SLU_DN, SLU_D, SLU_GE);

/* set options that control behavior of solver to default parameters */
set_default_options(&options);
options.ColPerm = NATURAL;

/* Initialize the statistics variables. */
StatInit(&stat);
/* factor input matrix and solve the first right-hand-side matrix */
dgssv(&options, &A, perm_c, perm_r, &L, &U, &B1, &stat, &info);

printf("\nsolution matrix B1:\n");
dp = (double *) (((NCformat *)B1.Store)->nzval);
printf("    i    rhs[i]     expected\n");
for (i=0; i<M; i++)
    printf("%5d   %7.4lf     %7.4lf\n", i, dp[i], 1.0);
printf("Factor time = %8.2e sec\n", stat.utime[FACT]);
printf("Solve time   = %8.2e sec\n\n", stat.utime[SOLVE]);

/* solve the second right-hand-side matrix */
dgstrs(trans, &L, &U, perm_c, perm_r, &B2, &stat, &info);

printf("solution matrix B2:\n");
dp = (double *) (((NCformat *)B2.Store)->nzval);
printf("    i    rhs[i]     expected\n");
for (i=0; i<M; i++)
    printf("%5d   %7.4lf     %7.4lf\n", i, dp[i], 0.3);
printf("Solve time   = %8.2e sec\n", stat.utime[SOLVE]);
StatFree(&stat);
Destroy_CompCol_Matrix(&A);
Destroy_SuperMatrix_Store(&B1);
Destroy_SuperMatrix_Store(&B2);
Destroy_SuperNode_Matrix(&L);
Destroy_CompCol_Matrix(&U);```
Running the above example:

```bash
code -dalign simple.c -xlic_lib=sunperf
```

Example code calling SuperLU simple driver to factor a general unsymmetric matrix and solve two right-hand-side matrices

Solution matrix B1:

<table>
<thead>
<tr>
<th>i</th>
<th>rhs[i]</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>3</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>4</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Factor time  = 5.43e-02 sec
Solve time   = 6.76e-03 sec

Solution matrix B2:

<table>
<thead>
<tr>
<th>i</th>
<th>rhs[i]</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3000</td>
<td>0.3000</td>
</tr>
<tr>
<td>1</td>
<td>0.3000</td>
<td>0.3000</td>
</tr>
<tr>
<td>2</td>
<td>0.3000</td>
<td>0.3000</td>
</tr>
<tr>
<td>3</td>
<td>0.3000</td>
<td>0.3000</td>
</tr>
<tr>
<td>4</td>
<td>0.3000</td>
<td>0.3000</td>
</tr>
</tbody>
</table>

Solve time   = 6.76e-03 sec

---

**CODE EXAMPLE 6-7** SuperLU Expert Driver

```c
#include <stdio.h>
#include <sunperf.h>

#define M    5
#define N    5
#define NRHS 1

int main(int argc, char *argv[])
{
    SuperMatrix A, L, U, B, X;
    int      perm_r[M];  /* row permutations from partial pivoting */
    int      perm_c[N];  /* column permutation vector */
    int      etree[N];   /* elimination tree */
    double   ferr[NRHS]; /* estimated forward error bound */
    double   berr[NRHS]; /* component-wise relative backward error */
    double   C[N], R[M]; /* column and row scale factors */
    double   rpg, rcond;
    char      equed[1];  /* Specifies the form of equilibration that was done */
```
```c
double *work, *dp; /* user-supplied workspace */
int lwork = 0; /* 0 for workspace to be allocated by system malloc */
int info, i;
superlu_options_t options;
SuperLUStat_t stat;
mem_usage_t mem_usage;

printf("Example code calling SuperLU expert driver\n\n");

/* the matrix in Harwell-Boeing format. */
int m = M;
int n = M;
int nnz = 12;
/* nonzeros of A, column-wise */
double a[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
             7.0, 8.0, 9.0, 10.0, 11.0, 12.0};
/* row index of nonzeros */
int asub[] = {0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4};
/* column pointers */
int xa[] = {0, 3, 6, 8, 10, 12};
int nrhs = NRHS;
double rhs[] = {17.0, 6.0, 13.0, 21.0, 21.0};

/* Create Matrix A in the format expected by SuperLU */
dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, SLU_NC, SLU_D, SLU_GE);

/* right-hand-side matrix B */
dCreate_Dense_Matrix(&B, m, nrhs, rhs, m, SLU_DN, SLU_D, SLU_GE);

/* solution matrix X */
dCreate_Dense_Matrix(&X, m, nrhs, rhs, m, SLU_DN, SLU_D, SLU_GE);
set_default_options(&options);
options.ColPerm = NATURAL;

/* Initialize the statistics variables. */
StatInit(&stat);
dgssvx(&options, &A, perm_c, perm_r, etree, equed, R, C, &L, &U, work, lwork,
       &B, &X, &rpg, &rcond, ferr, berr, &mem_usage, &stat, &info);
dp = (double *) (((NCformat *)X.Store)->nzval);
printf(" i rhs[i] expected\n");
for (i=0; i<M; i++)
    printf("%5d %7.4lf %7.4lf\n",
           i, dp[i], 1.0);
printf("Factor time = %8.2e sec\n", stat.utime[FACT]);
printf("Solve time = %8.2e sec\n", stat.utime[SOLVE]);
```
6.4.2 Calling from Fortran

The simplest way to call SuperLU from Fortran is through the SPSOLVE interface. SuperLU can be selected to solve an unsymmetric coefficient matrix through input argument MTXTYP of routine DGSSIN(), which is the initialization routine in SPSOLVE. The same argument also exists in the one-call interface routine DGSSFS(). Valid options for MTXTYP are listed in the following table. To invoke SuperLU, select ‘s0’ or ‘S0’ as matrix type. Since SPSOLVE is Fortran-based, all column and row indices associated with the input matrix should be one-based. However, if SuperLU is invoked through DGSSIN() or DGSSFS() (by setting MTXTYP = ‘s0’ or ‘S0’), these indices must be zero-based.

<table>
<thead>
<tr>
<th>Option</th>
<th>Type of Matrix</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘sp’ or ‘SP’</td>
<td>symmetric structure, positive-definite values</td>
<td>SPSOLVE</td>
</tr>
<tr>
<td>‘ss’ or ‘SS’</td>
<td>symmetric structure, symmetric values</td>
<td>SPSOLVE</td>
</tr>
</tbody>
</table>

Running the above example:

```c
StatFree(&stat);
Destroy_Complex_Column_Matrix(&A);
Destroy_SuperMatrix_Store(&B);
Destroy_SuperNode_Matrix(&L);
Destroy_Complex_Column_Matrix(&U);

my_system%
c -dalign expert.c -xlic_lib=sunperf
my_system%
a.out
Example code calling SuperLU expert driver

<table>
<thead>
<tr>
<th>i</th>
<th>rhs[i]</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>3</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>4</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
Factor time = 1.25e-03 sec
Solve time  = 1.70e-04 sec
```
A call to routine DGSSOR() must follow DGSSIN() to perform fill-reduce ordering and symbolic factorization. A character argument (ORDMTHD) is used to select the desired ordering method. This argument also exists in the one-call interface routine DGSSFS(). Valid ordering methods for SPSOLVE and SuperLU are listed in the following table. The user may also provide a particular ordering to the solver by calling DGSSUO() in place of DGSSOR(). The input permutation array must be zero-based.

### TABLE 6-6  Matrix Type Options for DGSSIN() and DGSSFS()

<table>
<thead>
<tr>
<th>Option</th>
<th>Type of Matrix</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘su’ or ‘SU’</td>
<td>symmetric structure, unsymmetric values</td>
<td>SPSOLVE</td>
</tr>
<tr>
<td>‘uu’ or ‘UU’</td>
<td>unsymmetric structure, unsymmetric values</td>
<td>SPSOLVE</td>
</tr>
<tr>
<td>‘s0’ or ‘S0’</td>
<td>unsymmetric structure, unsymmetric values</td>
<td>SuperLU</td>
</tr>
</tbody>
</table>

A call to routine DGSSOR() must follow DGSSIN() to perform fill-reduce ordering and symbolic factorization. A character argument (ORDMTHD) is used to select the desired ordering method. This argument also exists in the one-call interface routine DGSSFS(). Valid ordering methods for SPSOLVE and SuperLU are listed in the following table. The user may also provide a particular ordering to the solver by calling DGSSUO() in place of DGSSOR(). The input permutation array must be zero-based.

### TABLE 6-7  Matrix Ordering Options for DGSSOR() and DGSSFS()

<table>
<thead>
<tr>
<th>Option</th>
<th>Ordering Method</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘nat’ or ‘NAT’</td>
<td>natural ordering (no ordering)</td>
<td>SPSOLVE, SuperLU</td>
</tr>
<tr>
<td>‘mmd’ or ‘MMD’</td>
<td>minimum degree on $A^*A$ (default)</td>
<td>SPSOLVE, SuperLU</td>
</tr>
<tr>
<td>‘gnd’ or ‘GND’</td>
<td>general nested dissection</td>
<td>SPSOLVE</td>
</tr>
<tr>
<td>‘spm’ or ‘SPM’</td>
<td>Minimum degree ordering on $A^/+A$</td>
<td>SuperLU</td>
</tr>
<tr>
<td>‘sam’ or ‘SAM’</td>
<td>Approximate minimum degree column</td>
<td>SuperLU</td>
</tr>
</tbody>
</table>

As shown above, the general nested dissection method is not available in SuperLU. On the other hand, the minimum degree ordering on $A^/+A$ and approximate minimum degree column ordering are not available in SPSOLVE.

### 6.4.3 Examples

The following code examples show how SuperLU can be selected through the regular interface and the one-call interface of SPSOLVE to factorize and solve a general unsymmetric system of equations.
Invoking SuperLU through SPSOLVE Interface

```fortran
program SLU

! This program is an example driver that calls the regular interface of SPSOLVE to invoke SuperLU to factor and solve a general unsymmetric system.

implicit none
integer           neqns, ier, msglvl, outunt, ldrhs, nrhs, i
character         mtxtyp*2, pivot*1, ordmthd*3
double precision  handle(150)
integer           colstr(6), rowind(12)
double precision  values(12), rhs(5), xexpct(5)

! Sparse matrix structure and value arrays. Coefficient matrix
! Ax = b, (solve for x) where:

!          1.0  0.0  7.0  9.0  0.0   1.0   17.0
!          2.0  4.0  0.0  0.0  0.0   1.0    6.0
!         A =  0.0  5.0  8.0  0.0  0.0  x = 1.0   b = 13.0
!          0.0  0.0  0.0 10.0 11.0   1.0    21.0
!          3.0  6.0  0.0  0.0 12.0   1.0    21.0

! Array indices must be zero-based for calling SuperLU
! data colstr / 0, 3, 6, 8, 10, 12 /
data rowind / 0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4 /
data values / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0 /
data rhs    / 17.0, 6.0, 13.0, 21.0, 21.0 /
data xexpct / 1.0d0, 1.0d0, 1.0d0, 1.0d0, 1.0d0 /

! initialize solver
mtxtyp= 's0'
pivot = 'n'
neqns = 5
outunt = 6
msglvl = 0

! call regular interface
    call dgssin(mtxtyp, pivot, neqns, colstr, rowind, outunt, msglvl, &
             handle, ier)
    if ( ier .ne. 0 ) goto 110

! ordering and symbolic factorization
ordmthd = 'mmd'
call dgssor(ordmthd, handle, ier)
```

Chapter 6  Sparse Computation  6-71
Running the above example:

<table>
<thead>
<tr>
<th>i</th>
<th>rhs(i)</th>
<th>expected rhs(i)</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>1.00</td>
<td>-0.33E-15</td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
<td>1.00</td>
<td>0.22E-15</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>1.00</td>
<td>-0.11E-15</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>1.00</td>
<td>0.22E-15</td>
</tr>
</tbody>
</table>
CODE EXAMPLE 6-9   Invoking SuperLU through One-Call SPSOLVE Interface

program SLU_SINGLE
  c This program is an example driver that calls the regular interface of SPSOLVE
  c to invoke SuperLU to factor and solve a general unsymmetric system.

  implicit none
  integer           neqns, ier, msglvl, outunt, ldrhs, nrhs, i
  character         mtxtyp*2, pivot*1, ordmthd*3
  double precision  handle(150)
  integer           colstr(6), rowind(12)
  double precision  values(12), rhs(5), xexpct(5)

  c Sparse matrix structure and value arrays. Coefficient matrix
  c is a general unsymmetric sparse matrix.
  c \( Ax = b \), (solve for \( x \)) where:

  c
  \[
  \begin{bmatrix}
    1.0 & 0.0 & 7.0 & 9.0 & 0.0 & 1.0 & 17.0 \\
    2.0 & 4.0 & 0.0 & 0.0 & 0.0 & 1.0 & 6.0 \\
    0.0 & 5.0 & 8.0 & 0.0 & 0.0 & x & 13.0 \\
    0.0 & 0.0 & 0.0 & 10.0 & 11.0 & 1.0 & 21.0 \\
    3.0 & 6.0 & 0.0 & 0.0 & 12.0 & 1.0 & 21.0
  \end{bmatrix}
  \]
  c Array indices must be zero-based for calling SuperLU
  data colstr / 0, 3, 6, 8, 10, 12 /
  data rowind / 0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4 /
  data values / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0 /
  data rhs    / 17.0, 6.0, 13.0, 21.0, 21.0 /
  data xexpct / 1.0d0, 1.0d0, 1.0d0, 1.0d0, 1.0d0 /

  c initialize solver
  mtxtyp= ‘s0’
  pivot = ‘n’
  neqns = 5
  outunt = 6
  msglvl = 0
  ordmthd = ‘mmd’
  nrhs = 1
  ldrhs = 5

  c One-call routine of SPSOLVE
  call dgssfs (mtxtyp, pivot, neqns , colstr, rowind,
    & values, nrhs , rhs, ldrhs , ordmthd,
    & outunt, msglvl1, handle, ier)
  if ( ier .ne. 0 ) goto 110

  c deallocate sparse solver storage
  call dgssda ( handle, ier )

Chapter 6   Sparse Computation   6-73
6.5 References

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


Using Sun Performance Library Signal Processing Routines

The discrete Fourier transform (DFT) has always been an important analytical tool in many areas in science and engineering. However, it was not until the development of the fast Fourier transform (FFT) that the DFT became widely used. This is because the DFT requires $O(N^2)$ computations, while the FFT only requires $O(N \log_2 N)$ operations.

Sun Performance Library contains a set of routines that computes the FFT, related FFT operations, such as convolution and correlation, and trigonometric transforms.

This chapter is divided into the following three sections.

■ Forward and Inverse FFT Routines
■ Sine and Cosine Transforms
■ Convolution and Correlation

Each section includes examples that show how the routines might be used.

For information on the Fortran 95 and C interfaces and types of arguments used in each routine, see the section 3P man pages for the individual routines.

For example, to display the man page for the $\text{sfftc}$ routine, type

```
man -s 3P sfftc
```

Routine names must be lowercase. For an overview of the FFT routines, type

```
man -s 3P fft
```
# 7.1  Forward and Inverse FFT Routines

**TABLE 7-1** lists the names of the FFT routines and their calling sequence. Double precision routine names are in square brackets. See the individual man pages for detailed information on the data type and size of the arguments.

**TABLE 7-1  FFT Routines and Their Arguments**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linear Routines</strong></td>
<td></td>
</tr>
<tr>
<td>CFFTS [ZFFTD]</td>
<td>(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>SPFCTC [DFFTZ]</td>
<td>(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>CFFTSM [ZFFTD]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>SPFCTCM [DFFTZM]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>CFFTC [ZFFTZ]</td>
<td>(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>CFFTCM [ZFFTZM]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td><strong>Two-Dimensional Routines</strong></td>
<td></td>
</tr>
<tr>
<td>CFFTS2 [ZFFTD2]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>SPFCT2 [DFFTZ2]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>CFFTC2 [ZFFTZ2]</td>
<td>(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td><strong>Three-Dimensional Routines</strong></td>
<td></td>
</tr>
<tr>
<td>CFFTS3 [ZFFTD3]</td>
<td>(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>SPFCT3 [DFFTZ3]</td>
<td>(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
<tr>
<td>CFFTC3 [ZFFTZ3]</td>
<td>(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)</td>
</tr>
</tbody>
</table>

Sun Performance Library FFT routines use the following arguments.

- **OPT**: Flag indicating whether the routine is called to initialize or to compute the transform.
- **N1, N2, N3**: Problem dimensions for one, two, and three dimensional transforms.
■ **X**: Input array where X is of type COMPLEX if the routine is a complex-to-complex transform or a complex-to-real transform. X is of type REAL for a real-to-complex transform.

■ **Y**: Output array where Y is of type COMPLEX if the routine is a complex-to-complex transform or a real-to-complex transform. Y is of type REAL for a complex-to-real transform.

■ **LDX1, LDX2 and LDY1, LDY2**: LDX1 and LDX2 are the leading dimensions of the input array, and LDY1 and LDY2 are the leading dimensions of the output array. The FFT routines allow the output to overwrite the input, which is an in-place transform, or to be stored in a separate array apart from the input array, which is an out-of-place transform. In complex-to-complex transforms, the input data is of the same size as the output data. However, real-to-complex and complex-to-real transforms have different memory requirements for input and output data. Care must be taken to ensure that the input array is large enough to accommodate the transform results when computing an in-place transform.

■ **TRIGS**: Array containing the trigonometric weights.

■ **IFAC**: Array containing factors of the problem dimensions. The problem sizes are as follows:
  - Linear FFT: Problem size of dimension N1
  - Two-dimensional FFT: Problem size of dimensions N1 and N2
  - Three-dimensional FFT: Problem size of dimensions N1, N2, and N3

While N1, N2, and N3 can be of any size, a real-to-complex or a complex-to-real transform can be computed most efficiently when

\[ N1, N2, N3 = 2^p \times 3^q \times 4^r \times 5^s \]

and a complex-to-complex transform can be computed most efficiently when

\[ N1, N2, N3 = 2^p \times 3^q \times 4^r \times 5^s \times 7^t \times 11^u \times 13^v \]

where \( p, q, r, s, t, u, \) and \( v \) are integers and \( p, q, r, s, t, u, v \geq 0 \).

■ **WORK**: Workspace whose size depends on the routine and the number of threads that are being used to compute the transform if the routine is parallelized.

■ **LWORK**: Size of workspace. If LWORK is zero, the routine will allocate a workspace with the required size.

■ **SCALE**: A scalar with which the output is scaled. Occasionally in literature, the inverse transform is defined with a scaling factor of \( 1/N1 \) for one-dimensional transforms, \( 1/(N1 \times N2) \) for two-dimensional transforms, and \( 1/(N1 \times N2 \times N3) \) for three-dimensional transforms. In such case, the inverse transform is said to be normalized. If a normalized FFT is followed by its inverse FFT, the result is the original input data. The Sun Performance Library FFT routines are not normalized. However, normalization can be done easily by calling the inverse FFT routine with the appropriate scaling factor stored in SCALE.

■ **ERR**: A flag returning a nonzero value if an error is encountered in the routine and zero otherwise.
7.1.1 Linear FFT Routines

Linear FFT routines compute the FFT of real or complex data in one dimension only. The data can be one or more complex or real sequences. For a single sequence, the data is stored in a vector. If more than one sequence is being transformed, the sequences are stored column-wise in a two-dimensional array and a one-dimensional FFT is computed for each sequence along the column direction. The linear forward FFT routines compute

\[
X(k) = \sum_{n=0}^{N1-1} x(n) e^{-\frac{2\pi ink}{N1}}, \quad k = 0, \ldots, N1-1,
\]

where \( i = \sqrt{-1} \), or expressed in polar form,

\[
X(k) = \sum_{n=0}^{N1-1} x(n) \left( \cos\left(\frac{2\pi nk}{N1}\right) - i\sin\left(\frac{2\pi nk}{N1}\right) \right), \quad k = 0, \ldots, N1-1.
\]

The inverse FFT routines compute

\[
x(n) = \sum_{k=0}^{N1-1} X(k) e^{\frac{2\pi ink}{N1}}, \quad n = 0, \ldots, N1-1,
\]

or in polar form,

\[
x(n) = \sum_{n=0}^{N1-1} X(k) \left( \cos\left(\frac{2\pi nk}{N1}\right) + i\sin\left(\frac{2\pi nk}{N1}\right) \right), \quad n = 0, \ldots, N1-1.
\]

With the forward transform, if the input is one or more complex sequences of size \( N1 \), the result will be one or more complex sequences, each consisting of \( N1 \) unrelated data points. However, if the input is one or more real sequences, each containing \( N1 \) real data points, the result will be one or more complex sequences that are conjugate symmetric. That is,

\[
X(k) = X^*(N1-k), \quad k = \frac{N1}{2} + 1, \ldots, N1-1.
\]
The imaginary part of $X(0)$ is always zero. If $N1$ is even, the imaginary part of $X(\frac{N1}{2})$ is also zero. Both zeros are stored explicitly. Because the second half of each sequence can be derived from the first half, only $\frac{N1}{2} + 1$ complex data points are computed and stored in the output array. Here and elsewhere in this chapter, integer division is rounded down.

With the inverse transform, if an $N1$-point complex-to-complex transform is being computed, then $N1$ unrelated data points are expected in each input sequence and $N1$ data points will be returned in the output array. However, if an $N1$-point complex-to-real transform is being computed, only the first $\frac{N1}{2} + 1$ complex data points of each conjugate symmetric input sequence are expected in the input, and the routine will return $N1$ real data points in each output sequence.

For each value of $N1$, either the forward or the inverse routine must be called to compute the factors of $N1$ and the trigonometric weights associated with those factors before computing the actual FFT. The factors and trigonometric weights can be reused in subsequent transforms as long as $N1$ remains unchanged.

TABLE 7.2 lists the single precision linear FFT routines and their purposes. For routines that have two-dimensional arrays as input and output, TABLE 7.2 also lists the leading dimension requirements. The same information applies to the corresponding double precision routines except that their data types are double precision and double complex. See TABLE 7.2 for the mapping. See the individual man pages for a complete description of the routines and their arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Size and Type of Input</th>
<th>Size and Type of Output</th>
<th>Leading Dimension Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In-place</td>
</tr>
<tr>
<td>SFFTC</td>
<td>OPT = 0 initialization</td>
<td>$N1$, Real</td>
<td>$\frac{N1}{2} + 1$, Complex</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPT = -1 real-to-complex forward linear FFT of a single vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFFTC</td>
<td>OPT = 0 initialization</td>
<td>$\frac{N1}{2} + 1$, Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPT = 1 complex-to-real inverse linear FFT of a single vector</td>
<td>$\frac{N1}{2} + 1$, Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CFFT</td>
<td>OPT = 0 initialization</td>
<td>$N1$, Complex</td>
<td>$N1$, Complex</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Purpose</td>
<td>Size and Type of Input</td>
<td>Size and Type of Output</td>
<td>Leading Dimension Requirements</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------</td>
<td>------------------------</td>
<td>-------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>SFFTCM OPT = 0 initialization</td>
<td>N1 × M, Real, N1 × M, Complex</td>
<td>LDX1 = 2 × LDY1</td>
<td>N1, Complex</td>
<td>LDY1 ≥ N1</td>
</tr>
<tr>
<td>CFFTS M OPT = 0 initialization</td>
<td>(N1/2 + 1) × M, Complex</td>
<td>LDX1 ≥ N1/2 + 1</td>
<td>LDY1 = 2 × LDX1</td>
<td>LDY1 ≥ N1</td>
</tr>
<tr>
<td>CFFT CM OPT = 0 initialization</td>
<td>N1 × M, Complex</td>
<td>LDX1 ≥ N1</td>
<td>LDY1 ≥ N1</td>
<td>LDY1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td>OPT = -1 complex-to-complex forward linear FFT of M vectors</td>
<td>N1 × M, Complex</td>
<td>LDY1 ≥ N1</td>
<td>LDY1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td>OPT = 1 complex-to-complex inverse linear FFT of M vectors</td>
<td>N1 × M, Complex</td>
<td>LDY1 ≥ N1</td>
<td>LDY1 ≥ N1</td>
</tr>
</tbody>
</table>

**TABLE 7-2 Notes.**
- LDX1 is the leading dimension of the input array.
- LDY1 is the leading dimension of the output array.
- N1 is the first dimension of the FFT problem.
- N2 is the second dimension of the FFT problem.
- When calling routines with OPT = 0 to initialize the routine, the only error checking that is done is to determine if N1 < 0.

**CODE EXAMPLE 7-1** shows how to compute the linear real-to-complex and complex-to-real FFT of a set of sequences.

**CODE EXAMPLE 7-1**  Linear Real-to-Complex FFT and Complex-to-Real FFT

my_system% cat testscm.f

```fortran
PROGRAM TESTSCM
IMPLICIT NONE
INTEGER :: LW, IERR, I, J, K, LDX, LDC
```

7-82 Sun Performance Library User’s Guide • March 2009
```fortran
INTEGER, PARAMETER :: N1 = 3, N2 = 2, LDZ = N1,
$     LDC = N1, LDX = 2*LDC
INTEGER, DIMENSION(:) :: IFAC(128)
REAL :: SCALE
REAL, PARAMETER :: ONE = 1.0
REAL, DIMENSION(:) :: SW(N1), TRIGS(2*N1)
REAL, DIMENSION(0:LDX-1, 0:N2-1) :: X, V, Y
COMPLEX, DIMENSION(0:LDZ-1, 0:N2-1) :: Z
* workspace size
LW = N1
SCALE = ONE/N1
WRITE(*,*)
$ 'Linear complex-to-real and real-to-complex FFT of a sequence'
WRITE(*,*)
X = RESHAPE(SOURCE = (/0.1, 0.2, 0.3, 0.0, 0.0, 0.0, 7.0, 8.0, 9.0,
$    0.0, 0.0, 0.0/), SHAPE=(/6,2/))
V = X
WRITE(*,*) 'X = '
DO I = 0, N1-1
    WRITE(*,'(2(F4.1,2x))') (X(I,J), J = 0, N2-1)
END DO
WRITE(*,*)
* initialize trig table and compute factors of N1
CALL SFFTCM(0, N1, N2, ONE, X, LDX, Z, LDZ, TRIGS, IFAC,
$     SW, LW, IERR)
IF (IERR .NE. 0) THEN
    PRINT*, 'ROUTINE RETURN WITH ERROR CODE = ', IERR
    STOP
END IF
* Compute out-of-place forward linear FFT.
* Let FFT routine allocate memory.
CALL SFFTCM(-1, N1, N2, ONE, X, LDX, Z, LDZ, TRIGS, IFAC,
$     SW, LW, IERR)
IF (IERR .NE. 0) THEN
    PRINT*, 'ROUTINE RETURN WITH ERROR CODE = ', IERR
    STOP
END IF
WRITE(*,*) 'out-of-place forward FFT of X:
WRITE(*,*) 'Z ='
DO I = 0, N1/2
    WRITE(*,'(2(A1, F4.1,A1,F4.1,A1,2x))') ('(',REAL(Z(I,J)),
$    ',',AIMAG(Z(I,J)),')', J = 0, N2-1)
END DO
WRITE(*,*)
* Compute in-place forward linear FFT.
```

**CODE EXAMPLE 7-1**  
Linear Real-to-Complex FFT and Complex-to-Real FFT (Continued)
* X must be large enough to store N1/2+1 complex values
   CALL SFPTCM(-1, N1, N2, ONE, X, LDX, X, LDC, TRIGS, IFAC, $  SW, LW, IERR)
   IF (IERR .NE. 0) THEN
      PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
      STOP
   END IF
   WRITE(*,*) 'in-place forward FFT of X:'
   CALL PRINT_REAL_AS_COMPLEX(N1/2+1, N2, 1, X, LDC, N2)
   WRITE(*,*)
* Compute out-of-place inverse linear FFT.
   CALL CFPTSM(1, N1, N2, SCALE, Z, LDZ, X, LDX, TRIGS, IFAC, $  SW, LW, IERR)
   IF (IERR .NE. 0) THEN
      PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
      STOP
   END IF
   WRITE(*,*) 'out-of-place inverse FFT of Z:'
   DO I = 0, N1-1
      WRITE(*,'(2(F4.1,2X))') (X(I,J), J = 0, N2-1)
   END DO
   WRITE(*,*)
* Compute in-place inverse linear FFT.
   CALL CFPTSM(1, N1, N2, SCALE, Z, LDZ, Z, LDZ*2, TRIGS, IFAC, $  0, SW, LW, IERR)
   IF (IERR .NE. 0) THEN
      PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
      STOP
   END IF
   WRITE(*,*) 'in-place inverse FFT of Z:'
   CALL PRINT_COMPLEX_AS_REAL(N1, N2, 1, Z, LDZ*2, N2)
   WRITE(*,*)
END PROGRAM TESTSCM

SUBROUTINE PRINT_COMPLEX_AS_REAL(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K
REAL A(LD1, LD2, *)
DO K = 1, N3
   DO I = 1, N1
      WRITE(*,'(5(F4.1,2X))') (A(I,J,K), J = 1, N2)
   END DO
   WRITE(*,*)
END DO
WRITE(*,*)
END

SUBROUTINE PRINT_REAL_AS_COMPLEX(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K

CODE EXAMPLE 7-1  Linear Real-to-Complex FFT and Complex-to-Real FFT (Continued)
CODE EXAMPLE 7-1 Notes:

The forward FFT of X is actually

\[
\begin{array}{cc}
(0.6, 0.0) & (24.0, 0.0) \\
(-0.2, 0.1) & (-1.5, 0.9) \\
(-0.2, -0.1) & (-1.5, -0.9)
\end{array}
\]

Because of symmetry, Z(2) is the complex conjugate of Z(1), and therefore only the first two \( \frac{N}{2} + 1 = 2 \) complex values are stored. For the in-place forward transform, `SFFTCM` is called with real array X as the input and output. Because `SFFTCM` expects the output array to be of type `COMPLEX`, the leading dimension of X as an output
array must be as if X were complex. Since the leading dimension of real array X is $LDX = 2 \times LDC$, the leading dimension of X as a complex output array must be $LDC$. Similarly, in the in-place inverse transform, CFFTSM is called with complex array Z as the input and output. Because CFFTSM expects the output array to be of type REAL, the leading dimension of Z as an output array must be as if Z were real. Since the leading dimension of complex array Z is $LDZ$, the leading dimension of Z as a real output array must be $LDZ \times 2$.

CODE EXAMPLE 7-2 shows how to compute the linear complex-to-complex FFT of a set of sequences.

**CODE EXAMPLE 7-2  Linear Complex-to-Complex FFT**

```fortran
my_system% cat testccm.f

PROGRAM TESTCCM
IMPLICIT NONE
INTEGER :: LDX1, LDY1, LW, IERR, I, J, K, LDZ1, NCPUS,$
           USING_THREADS, IFAC(128)
INTEGER, PARAMETER :: N1 = 3, N2 = 4, LDX1 = N1, LDZ1 = N1,$
           LDY1 = N1+2
REAL, PARAMETER :: ONE = 1.0, SCALE = ONE/N1
COMPLEX :: Z(0:LDZ1-1,0:N2-1), X(0:LDX1-1,0:N2-1),$
           Y(0:LDY1-1,0:N2-1)
REAL :: TRIGS(2*N1)
REAL, DIMENSION(:), ALLOCATABLE :: SW

* get number of threads
NCPUS = USING_THREADS()
* workspace size
LW = 2 * N1 * NCPUS
WRITE(*,*)'Linear complex-to-complex FFT of one or more sequences'
WRITE(*,*)
ALLOCATE(SW(LW))
X = RESHAPE(SOURCE =(/(.1,.2),(.3,.4),(.5,.6),(.7,.8),(.9,1.0),$
           (1.1,1.2),(1.3,1.4),(1.5,1.6),(1.7,1.8),(1.9,2.0),(2.1,2.2),$
           (1.2,2.0)/), SHAPE=(/LDX1,N2/))
Z = X
WRITE(*,*)'X = '
DO I = 0, N1-1
   WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))') ('(',REAL(X(I,J)),$
           ',',AIMAG(X(I,J)),')', J = 0, N2-1)
END DO
* intialize trig table and compute factors of N1
CALL CFFTCM(0, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC,$
           SW, LW, IERR)
IF (IERR .NE. 0) THEN
```

7-86  Sun Performance Library User's Guide  •  March 2009
* Compute out-of-place forward linear FFT.
* Let FFT routine allocate memory.
  CALL CPFTCM(-1, N1, N2, ONE, X, LDX1, Y, LDY1, TRIGS, IFAC,
             SW, 0, IERR)
  IF (IERR .NE. 0) THEN
      PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
      STOP
  END IF
  * Compute in-place forward linear FFT. LDZ1 must equal LDX1
  CALL CPFTCM(-1, N1, N2, ONE, Z, LDX1, Z, LDZ1, TRIGS,
             IFAC, SW, 0, IERR)
  WRITE(*,*) 'in-place forward FFT of X:'
  DO I = 0, N1-1
      WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))') ('(',REAL(Z(I,J)),
             ',',AIMAG(Z(I,J)),')', J = 0, N2-1)
  END DO
  WRITE(*,*)
  WRITE(*,**) 'out-of-place forward FFT of X:
  WRITE(*,**) 'Y ='
  DO I = 0, N1-1
      WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))') ('(',REAL(Y(I,J)),
             ',',AIMAG(Y(I,J)),')', J = 0, N2-1)
  END DO
  WRITE(*,**)
  * Compute in-place inverse linear FFT.
  CALL CPFTCM(1, N1, N2, SCALE, Y, LDY1, Y, LDY1, TRIGS, IFAC,
             SW, LW, IERR)
  IF (IERR .NE. 0) THEN
      PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
      STOP
  END IF
  WRITE(*,**) 'in-place inverse FFT of Y:'
  WRITE(*,**) 'Y ='
  DO I = 0, N1-1
      WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))') ('(',REAL(Y(I,J)),
             ',',AIMAG(Y(I,J)),')', J = 0, N2-1)
  END DO
  DEALLOCATE(SW)
END PROGRAM TESTCCM

my_system% f95 -dalign testccm.f -xlic_lib=sunperf
my_system% a.out
Linear complex-to-complex FFT of one or more sequences
7.1.2 Two-Dimensional FFT Routines

For the linear FFT routines, when the input is a two-dimensional array, the FFT is computed along one dimension only, namely, along the columns of the array. The two-dimensional FFT routines take a two-dimensional array as input and compute the FFT along both the column and row dimensions. Specifically, the forward two-dimensional FFT routines compute

\[
X_k = \frac{1}{N_2} \sum_{n=0}^{N_2-1} x(n) e^{-2\pi i jk/N_2}, \quad k = 0, \ldots, N_1-1, \quad j = 0, \ldots, N_2-1,
\]

and the inverse two-dimensional FFT routines compute

\[
x(n) = \frac{1}{N_1} \sum_{j=0}^{N_1-1} X(k,j) e^{2\pi i nk/N_1}, \quad n = 0, \ldots, N_2-1, \quad j = 0, \ldots, N_1-1.
\]

For both the forward and inverse two-dimensional transforms, a complex-to-complex transform where the input problem is \(N_1 \times N_2\) will yield a complex array that is also \(N_1 \times N_2\).

When computing a real-to-complex two-dimensional transform (forward FFT), if the real input array is of dimensions \(N_1 \times N_2\), the result will be a complex array of dimensions \(\left(\frac{N_1}{2} + 1\right) \times N_2\). Conversely, when computing a complex-to-real transform

\[
X =
\begin{pmatrix}
(0.1, 0.2) & (0.7, 0.8) & (1.3, 1.4) & (1.9, 2.0) \\
(0.3, 0.4) & (0.9, 1.0) & (1.5, 1.6) & (2.1, 2.2) \\
(0.5, 0.6) & (1.1, 1.2) & (1.7, 1.8) & (1.2, 2.0)
\end{pmatrix}
\]

in-place forward FFT of X:

\[
\begin{pmatrix}
(0.9, 1.2) & (2.7, 3.0) & (4.5, 4.8) & (5.2, 6.2) \\
(-0.5, -0.1) & (-0.5, -0.1) & (-0.5, -0.1) & (0.4, -0.9) \\
(-0.1, -0.5) & (-0.1, -0.5) & (-0.1, -0.5) & (0.1, 0.7)
\end{pmatrix}
\]

out-of-place forward FFT of X:

\[
Y =
\begin{pmatrix}
(0.1, 0.2) & (0.7, 0.8) & (1.3, 1.4) & (1.9, 2.0) \\
(0.3, 0.4) & (0.9, 1.0) & (1.5, 1.6) & (2.1, 2.2) \\
(0.5, 0.6) & (1.1, 1.2) & (1.7, 1.8) & (1.2, 2.0)
\end{pmatrix}
\]

in-place inverse FFT of Y:

\[
Y =
\begin{pmatrix}
(0.1, 0.2) & (0.7, 0.8) & (1.3, 1.4) & (1.9, 2.0) \\
(0.3, 0.4) & (0.9, 1.0) & (1.5, 1.6) & (2.1, 2.2) \\
(0.5, 0.6) & (1.1, 1.2) & (1.7, 1.8) & (1.2, 2.0)
\end{pmatrix}
\]
(inverse FFT) of dimensions \( N_1 \times N_2 \), an \( (\frac{N_1}{2} + 1) \times N_2 \) complex array is required as input. As with the real-to-complex and complex-to-real linear FFT, because of conjugate symmetry, only the first \( \frac{N_1}{2} + 1 \) complex data points need to be stored in the input or output array along the first dimension. The complex subarray \( X(\frac{N_1}{2} + 1: N_1 - 1, :) \) can be obtained from \( X(0: \frac{N_1}{2}, :) \) as follows:

\[
X(k, n) = X^r(N_1 - k, n),
\]

\[
k = \frac{N_1}{2} + 1, \ldots, N_1 - 1
\]

\[
n = 0, \ldots, N_2 - 1
\]

To compute a two-dimensional transform, an FFT routine must be called twice. One call initializes the routine and the second call actually computes the transform. The initialization includes computing the factors of \( N_1 \) and \( N_2 \) and the trigonometric weights associated with those factors. In subsequent forward or inverse transforms, initialization is not necessary as long as \( N_1 \) and \( N_2 \) remain unchanged.

**IMPORTANT:** Upon returning from a two-dimensional FFT routine, \( Y(0 : N - 1, :) \) contains the transform results and the original contents of \( Y(N : LDY-1, :) \) is overwritten. Here, \( N = N_1 \) in the complex-to-complex and complex-to-real transforms and \( N = \frac{N_1}{2} + 1 \) in the real-to-complex transform.

**TABLE 7-3** lists the single precision two-dimensional FFT routines and their purposes. The same information applies to the corresponding double precision routines except that their data types are double precision and double complex. See **TABLE 7-3** for the mapping. Refer to the individual man pages for a complete description of the routines and their arguments.

**TABLE 7-3** Single Precision Two-Dimensional FFT Routines

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Size, Type of Input</th>
<th>Size, Type of Output</th>
<th>Leading Dimension Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFFTC2</td>
<td>OPT = 0 initialization</td>
<td>( N_1 \times N_2 ), Real</td>
<td>( (\frac{N_1}{2} + 1) \times N_2 ), Complex</td>
<td>In-place: ( LDX1 = 2 \times LDY1 ), Out-of-Place: ( LDX1 \geq N_1, LDY1 \geq \frac{N_1}{2} + 1 )</td>
</tr>
<tr>
<td></td>
<td>OPT = -1 real-to-complex forward two-dimensional FFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CFFTS2</td>
<td>OPT = 0 initialization</td>
<td>( (\frac{N_1}{2} + 1) \times N_2 ), Complex</td>
<td>( N_1 \times N_2 ), Real</td>
<td>In-place: ( LDX1 \geq \frac{N_1}{2} + 1 ), Out-of-Place: ( LDX1 \geq \frac{N_1}{2} + 1 )</td>
</tr>
<tr>
<td></td>
<td>OPT = 1 complex-to-real inverse two-dimensional FFT</td>
<td></td>
<td></td>
<td>In-place: ( LDY1 = 2 \times LDX1 ), Out-of-Place: ( LDY1 \geq 2 \times LDX1 ), ( LDY1 ) is even</td>
</tr>
</tbody>
</table>
TABLE 7-3 Notes:

■ LDX1 is the leading dimension of the input array.
■ LDY1 is the leading dimension of the output array.
■ N1 is the first dimension of the FFT problem.
■ N2 is the second dimension of the FFT problem.
■ When calling routines with OPT = 0 to initialize the routine, the only error checking that is done is to determine if N1, N2 < 0.

The following example shows how to compute a two-dimensional real-to-complex FFT and complex-to-real FFT of a two-dimensional array.

CODE EXAMPLE 7-3

Two-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Two-Dimensional Array

```
my_system% cat testsc2.f
PROGRAM TESTSC2
IMPLICIT NONE
INTEGER, PARAMETER :: N1 = 3, N2 = 4, LDX1 = N1,
$         LDY1 = N1/2+1, LDR1 = 2*(N1/2+1)
INTEGER LW, IERR, I, J, K, IFAC(128*2)
REAL, PARAMETER :: ONE = 1.0, SCALE = ONE/(N1*N2)
REAL :: V(LDR1,N2), X(LDX1, N2), Z(LDR1,N2),
$        SW(2*N2), TRIGS(2*(N1+N2))
COMPLEX :: Y(LDY1,N2)
WRITE(*,*) $'Two-dimensional complex-to-real and real-to-complex FFT'
WRITE(*,*) X = RESHAPE(SOURCE = (/1., .2, .3, .4, .5, .6, .7, .8,
$       2.0,1.0, 1.1, 1.2/), SHAPE=/(LDX1,N2/))
DO I = 1, N2
  V(1:N1,I) = X(1:N1,I)
END DO
```
WRITE(*,*) 'X ='
DO I = 1, N1
WRITE(*,'(5(F5.1,2X))') (X(I,J), J = 1, N2)
END DO
WRITE(*,*)

* Initialize trig table and get factors of N1, N2
CALL SPFTC2(0,N1,N2,ONE,X,LDX1,Y,LDY1,TRIGS,
$            IFAC,SW,0,IERR)

* Compute 2-dimensional out-of-place forward FFT.
* Let FFT routine allocate memory.
* cannot do an in-place transform in X because LDX1 < 2*(N1/2+1)
CALL SPFTC2(-1,N1,N2,ONE,X,LDX1,Y,LDY1,TRIGS,
$            IFAC,SW,0,IERR)
WRITE(*,*) 'out-of-place forward FFT of X:'
WRITE(*,*)'Y ='
DO I = 1, N1/2+1
WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))')('(',REAL(Y(I,J)),
$           ',',AIMAG(Y(I,J)),')', J = 1, N2)
END DO
WRITE(*,*)

* Compute 2-dimensional in-place forward FFT.
* Use workspace already allocated.
* V which is real array containing input data is also
* used to store complex results; as a complex array, its first
* leading dimension is LDR1/2.
CALL SPFTC2(-1,N1,N2,ONE,V,LDR1,V,LDR1/2,TRIGS,
$            IFAC,SW,LW,IERR)
WRITE(*,*) 'in-place forward FFT of X:'
CALL PRINT_REAL_AS_COMPLEX(N1/2+1, N2, 1, V, LDR1/2, N2)

* Compute 2-dimensional out-of-place inverse FFT.
* Leading dimension of Z must be even
CALL CFFTS2(1,N1,N2,SCALE,Y,LDY1,Z,LDR1,TRIGS,
$            IFAC,SW,0,IERR)
WRITE(*,*) 'out-of-place inverse FFT of Y:'
DO I = 1, N1
WRITE(*,'(5(F5.1,2X))') (Z(I,J), J = 1, N2)
END DO
WRITE(*,*)

* Compute 2-dimensional in-place inverse FFT.
* Y which is complex array containing input data is also
* used to store real results; as a real array, its first
* leading dimension is 2*LDY1.
CALL CFFTS2(1,N1,N2,SCALE,Y,LDY1,Y,2*LDY1,
$            TRIGS,IFAC,SW,0,IERR)

CODE EXAMPLE 7-3  Two-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Two-Dimensional Array  (Continued)
WRITE(*,*) 'in-place inverse FFT of Y:'
CALL PRINT_COMPLEX_AS_REAL(N1, N2, 1, Y, 2*LDY1, N2)
END PROGRAM TESTSC2

SUBROUTINE PRINT_COMPLEX_AS_REAL(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K
REAL A(LD1, LD2, *)
DO K = 1, N3
  DO I = 1, N1
    WRITE(*,'(5(F5.1,2X))') (A(I,J,K), J = 1, N2)
  END DO
  WRITE(*,*)
END DO
END

SUBROUTINE PRINT_REAL_AS_COMPLEX(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K
COMPLEX A(LD1, LD2, *)
DO K = 1, N3
  DO I = 1, N1
    WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))') ('(',REAL(A(I,J,K)),',',AIMAG(A(I,J,K)),''), J = 1, N2)
  END DO
  WRITE(*,*)
END DO
END

my_system% f95 -dalign testsc2.f -xlic_lib=sunperf
my_system% a.out

Two-dimensional complex-to-real and real-to-complex FFT

x =
0.1 0.4 0.7 1.0
0.2 0.5 0.8 1.1
0.3 0.6 2.0 1.2

out-of-place forward FFT of X:
Y =
( 8.9, 0.0) ( -2.9, 1.8) ( -0.7, 0.0) ( -2.9, -1.8)
( -1.2, 1.3) ( 0.5, -1.0) ( -0.5, 1.0) ( 0.5, -1.0)
in-place forward FFT of X:
( 8.9, 0.0) ( -2.9, 1.8) ( -0.7, 0.0) ( -2.9, -1.8)
( -1.2, 1.3) ( 0.5, -1.0) ( -0.5, 1.0) ( 0.5, -1.0)

out-of-place inverse FFT of Y:
0.1 0.4 0.7 1.0
0.2 0.5 0.8 1.1
0.3 0.6 2.0 1.2
in-place inverse FFT of Y:
Three-Dimensional FFT Routines

Sun Performance Library includes routines that compute three-dimensional FFT. In this case, the FFT is computed along all three dimensions of a three-dimensional array. The forward FFT computes

\[
X(k, n, m) = \sum_{h=0}^{N_3-1} \sum_{l=0}^{N_2-1} \sum_{j=0}^{N_1-1} x(j, l, h) e^{2\pi i mh/N_3} e^{2\pi i ln/N_2} e^{2\pi i jk/N_1},
\]

where

\[
k = 0, ..., N_1 - 1
\]
\[
n = 0, ..., N_2 - 1
\]
\[
m = 0, ..., N_3 - 1
\]

and the inverse FFT computes

\[
x(j, l, h) = \sum_{m=0}^{N_3-1} \sum_{n=0}^{N_2-1} \sum_{k=0}^{N_1-1} X(k, n, m) e^{2\pi i mh/N_3} e^{2\pi i ln/N_2} e^{2\pi i jk/N_1},
\]

where

\[
j = 0, ..., N_1 - 1
\]
\[
l = 0, ..., N_2 - 1
\]
\[
h = 0, ..., N_3 - 1
\]

In the complex-to-complex transform, if the input problem is \(N_1 \times N_2 \times N_3\), a three-dimensional transform will yield a complex array that is also \(N_1 \times N_2 \times N_3\). When computing a real-to-complex three-dimensional transform, if the real input array is of dimensions \(N_1 \times N_2 \times N_3\), the result will be a complex array of dimensions \((\frac{N_3}{2} + 1) \times N_2 \times N_3\). Conversely, when computing a complex-to-real FFT of dimensions \(N_1 \times N_2 \times N_3\), an \((\frac{N_3}{2} + 1) \times N_2 \times N_3\) complex array is required as input. As with the real-to-complex and complex-to-real linear FFT, because of conjugate symmetry, only the first \(\frac{N_3}{2} + 1\) complex data points need to be stored along the first dimension. The complex subarray \(X(\frac{N_3}{2} + 1: N_1 - 1, :, :)\) can be obtained from \(X(0: \frac{N_3}{2}, :, :)\) as follows:

| 0.1 | 0.4 | 0.7 | 1.0 |
| 0.2 | 0.5 | 0.8 | 1.1 |
| 0.3 | 0.6 | 2.0 | 1.2 |

**CODE EXAMPLE 7-3** Two-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Two-Dimensional Array (Continued)
To compute a three-dimensional transform, an FFT routine must be called twice: Once to initialize and once more to actually compute the transform. The initialization includes computing the factors of $N_1$, $N_2$, and $N_3$ and the trigonometric weights associated with those factors. In subsequent forward or inverse transforms, initialization is not necessary as long as $N_1$, $N_2$, and $N_3$ remain unchanged.

**IMPORTANT:** Upon returning from a three-dimensional FFT routine, $Y(0:N-1,:,:)$ contains the transform results and the original contents of $Y(N:\text{LDY1-1},,:,:)$. Here, $N = N_1$ in the complex-to-complex and complex-to-real transforms and $N = \frac{N_1}{2} + 1$ in the real-to-complex transform.

**TABLE 7-4** lists the single precision three-dimensional FFT routines and their purposes. The same information applies to the corresponding double precision routines except that their data types are double precision and double complex. See **TABLE 7-4** for the mapping. See the individual man pages for a complete description of the routines and their arguments.

**TABLE 7-4** Single Precision Three-Dimensional FFT Routines

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Size, Type of Input</th>
<th>Size, Type of Output</th>
<th>Leading Dimension Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFFTC3</td>
<td>OPT = 0 initialization</td>
<td>$N_1 \times N_2 \times N_3$, Real</td>
<td>$(\frac{N_1}{2} + 1) \times N_2 \times N_3$, Complex</td>
<td>LDX1 = 2 × LDY1, LDX1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td>OPT = -1 real-to-complex forward three-dimensional FFT</td>
<td></td>
<td></td>
<td>LDX2 ≥ N2, LDY1 ≥ $\frac{N_1}{2} + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY2 = LDX2, LDY2 ≥ N2</td>
</tr>
<tr>
<td>CFFTS3</td>
<td>OPT = 0 initialization</td>
<td>$(\frac{N_1}{2} + 1) \times N_2 \times N_3$, Complex</td>
<td>$N_1 \times N_2 \times N_3$, Real</td>
<td>LDX1 ≥ $\frac{N_1}{2} + 1$</td>
</tr>
<tr>
<td></td>
<td>OPT = 1 complex-to-real inverse three-dimensional FFT</td>
<td></td>
<td></td>
<td>LDX2 ≥ N2, LDY1 ≥ $\frac{N_1}{2} + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY2 = LDX2, LDY1 is even, LDY2 ≥ N2</td>
</tr>
</tbody>
</table>
TABLE 7-4 Single Precision Three-Dimensional FFT Routines (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Size, Type of Input</th>
<th>Size, Type of Output</th>
<th>Leading Dimension Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In-place</td>
</tr>
<tr>
<td>CFFTC3</td>
<td>OPT = 0 initialization</td>
<td>N1 × N2 × N3, Complex</td>
<td>N1 × N2 × N3, Complex</td>
<td>LDX1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td>OPT = -1 complex-to-complex forward three-dimensional FFT</td>
<td>N1 × N2 × N3, Complex</td>
<td>N1 × N2 × N3, Complex</td>
<td>LDX1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDX2 ≥ N2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY1=LDX1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY2=LDX2</td>
</tr>
<tr>
<td></td>
<td>OPT = 1 complex-to-complex inverse three-dimensional FFT</td>
<td>N1 × N2 × N3, Complex</td>
<td>N1 × N2 × N3, Complex</td>
<td>LDX1 ≥ N1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDX2 ≥ N2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY1=LDX1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDY2=LDX2</td>
</tr>
</tbody>
</table>

TABLE 7-4 Notes:
- LDX1 is first leading dimension of input array.
- LDX2 is the second leading dimension of the input array.
- LDY1 is the first leading dimension of the output array.
- LDY2 is the second leading dimension of the output array.
- N1 is the first dimension of the FFT problem.
- N2 is the second dimension of the FFT problem.
- N3 is the third dimension of the FFT problem.
- When calling routines with OPT = 0 to initialize the routine, the only error checking that is done is to determine if N1, N2, N3 < 0.

CODE EXAMPLE 7-4 shows how to compute the three-dimensional real-to-complex FFT and complex-to-real FFT of a three-dimensional array.

CODE EXAMPLE 7-4 Three-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Three-Dimensional Array

```fortran
my_system% cat testsc3.f
PROGRAM TESTSC3
IMPLICIT NONE
INTEGER LW, NCPUS, IERR, I, J, K, USING_THREADS, IFAC(128*3)
INTEGER, PARAMETER :: N1 = 3, N2 = 4, N3 = 2, LDX1 = N1,$
$                      LDX2 = N2, LDY1 = N1/2+1, LDY2 = N2,$
$                      LDR1 = 2*(N1/2+1), LDR2 = N2$,
REAL, PARAMETER :: ONE = 1.0, SCALE = ONE/(N1*N2*N3)
```
REAL :: V(LDR1,LDR2,N3), X(LDX1,LDX2,N3), Z(LDR1,LDR2,N3),
        TRIGS(2*(N1+N2+N3))
REAL, DIMENSION(:), ALLOCATABLE :: SW
COMPLEX :: Y(LDY1,LDY2,N3)
WRITE(*,*)
$'Three-dimensional complex-to-real and real-to-complex FFT'
WRITE(*,*)
* get number of threads
NCPUS = USING_THREADS()
* compute workspace size required
LW = (MAX(MAX(N1,2*N2),2*N3) + 16*N3) * NCPUS
ALLOCATE(SW(LW))
X = RESHAPE(SOURCE =
    (/ .1, .2, .3, .4, .5, .6, .7, .8, .9,1.0,1.1,1.2,
     4.1,1.2,2.3,3.4,6.5,1.6,2.7,4.8,7.9,1.0,3.1,2.2/),
    SHAPE=(/LDX1,LDX2,N3/))
V = RESHAPE(SOURCE =
    (/,.1,.2,.3,.0,.4,.5,.6,.0,.7,.8,.9,.0,.1,.0,.1,.1,.2,.0,,
     4.1,.1,.2,.2,.3,.0,.3,.4,.6,.5,.1,.6,.0,.2,.7,.4,.8,.7,.9,.0,,
     1.0,.3,.1,.2,.2,.0./), SHAPE=(/LDR1,LDR2,N3/))
WRITE(*,*) 'X ='
DO K = 1, N3
  DO I = 1, N1
    WRITE(*,'(5(F5.1,2X))') (X(I,J,K), J = 1, N2)
  END DO
END DO
WRITE(*,*)
* Initialize trig table and get factors of N1, N2 and N3
CALL SFFTC3(0,N1,N2,N3,ONE,X,LDX1,LDX2,Y,LDY1,LDY2,TRIGS,
    $       IFAC,SW,0,IERR)
* Compute 3-dimensional out-of-place forward FFT.
* Let FFT routine allocate memory.
* cannot do an in-place transform because LDX1 < 2*(N1/2+1)
CALL SFFTC3(-1,N1,N2,N3,ONE,X,LDX1,LDX2,Y,LDY1,LDY2,TRIGS,
    $       IFAC,SW,0,IERR)
WRITE(*,*) 'out-of-place forward FFT of X:'
WRITE(*,*)'Y ='
DO K = 1, N3
  DO I = 1, N1/2+1
    WRITE(*,'(5(A1, F5.1,A1,F5.1,A1,2X))')('(',REAL(Y(I,J,K)),
        $      ',',AIMAG(Y(I,J,K)),')', J = 1, N2)
  END DO
END DO
WRITE(*,*)
END
Code Example 7-4  Three-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Three-Dimensional Array (Continued)

* Compute 3-dimensional in-place forward FFT.
* Use workspace already allocated.
* V which is real array containing input data is also
* used to store complex results; as a complex array, its first
* leading dimension is LDR1/2.
  CALL SFPTC3(-1,N1,N2,N3,ONE,V,LDR1,LDR2,V,LDR1/2,LDR2,TRIGS,
            $   IFAC,SW,LW,IERR)
  WRITE(*,'(A)') 'in-place forward FFT of X:'
  CALL PRINT_REAL_AS_COMPLEX(N1/2+1, N2, N3, V, LDR1/2, LDR2)

* Compute 3-dimensional out-of-place inverse FFT.
* First leading dimension of Z (LDR1) must be even
  CALL CFPTS3(1,N1,N2,N3,SCALE,Y,LDY1,LDY2,Z,LDR1,LDR2,TRIGS,
            $   IFAC,SW,0,IERR)
  WRITE(*,'(A)') 'out-of-place inverse FFT of Y:'
  DO K = 1, N3
    DO I = 1, N1
      WRITE(*,'(5(F5.1,2X))') (Z(I,J,K), J = 1, N2)
    END DO
  END DO
  WRITE(*,'(A)')
  DEALLOCATE(SW)
END PROGRAM TESTSC3

SUBROUTINE PRINT_REAL_AS_COMPLEX(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K
REAL A(LD1, LD2, *)
  DO K = 1, N3
    DO I = 1, N1
      WRITE(*,'(5(F5.1,2X))') (A(I,J,K), J = 1, N2)
    END DO
  END DO
END

SUBROUTINE PRINT_COMPLEX_AS_REAL(N1, N2, N3, A, LD1, LD2)
INTEGER N1, N2, N3, I, J, K
REAL A(LD1, LD2, *)
  DO K = 1, N3
    DO I = 1, N1
      WRITE(*,'(5(F5.1,2X))') (A(I,J,K), J = 1, N2)
    END DO
  END DO
END

Chapter 7  Using Sun Performance Library Signal Processing Routines  7-97
Three-dimensional complex-to-real and real-to-complex FFT

X =
0.1 0.4 0.7 1.0
0.2 0.5 0.8 1.1
0.3 0.6 0.9 1.2
4.1 3.4 2.7 1.0
1.2 6.5 4.8 3.1
2.3 1.6 7.9 2.2

out-of-place forward FFT of X:
Y =
( 48.6, 0.0) ( -9.6, -3.4) ( 3.4, 0.0) ( -9.6, 3.4)
( -4.2, -1.0) ( 2.5, -2.7) ( 1.0, 8.7) ( 9.5, -0.7)
(-33.0, 0.0) ( 6.0, 7.0) ( -7.0, 0.0) ( 6.0, -7.0)
( 3.0, 1.7) ( -2.5, 2.7) ( -1.0, -8.7) ( -9.5, 0.7)
in-place forward FFT of X:
( 48.6, 0.0) ( -9.6, -3.4) ( 3.4, 0.0) ( -9.6, 3.4)
( -4.2, -1.0) ( 2.5, -2.7) ( 1.0, 8.7) ( 9.5, -0.7)
(-33.0, 0.0) ( 6.0, 7.0) ( -7.0, 0.0) ( 6.0, -7.0)
( 3.0, 1.7) ( -2.5, 2.7) ( -1.0, -8.7) ( -9.5, 0.7)
out-of-place inverse FFT of Y:
0.1 0.4 0.7 1.0
0.2 0.5 0.8 1.1
0.3 0.6 0.9 1.2
4.1 3.4 2.7 1.0
1.2 6.5 4.8 3.1
2.3 1.6 7.9 2.2
in-place inverse FFT of Y:
0.1 0.4 0.7 1.0
0.2 0.5 0.8 1.1
0.3 0.6 0.9 1.2
4.1 3.4 2.7 1.0
1.2 6.5 4.8 3.1
2.3 1.6 7.9 2.2
7.1.4 Comments

When doing an in-place real-to-complex or complex-to-real transform, care must be taken to ensure the size of the input array is large enough to hold the results. For example, if the input is of type complex stored in a complex array with first leading dimension \( N \), then to use the same array to store the real results, its first leading dimension as a real output array would be \( 2 \times N \). Conversely, if the input is of type real stored in a real array with first leading dimension \( 2 \times N \), then to use the same array to store the complex results, its first leading dimension as a complex output array would be \( N \). Leading dimension requirements for in-place and out-of-place transforms can be found in TABLE 7-2, TABLE 7-3, and TABLE 7-4.

In the linear and multi-dimensional FFT, the transform between real and complex data through a real-to-complex or complex-to-real transform can be confusing because \( N \) real data points correspond to \( \frac{N}{2} + 1 \) complex data points. \( N \) real data points do map to \( N \) complex data points, but because there is conjugate symmetry in the complex data, only \( \frac{N}{2} + 1 \) data points need to be stored as input in the complex-to-real transform and as output in the real-to-complex transform. In the multi-dimensional FFT, symmetry exists along all the dimensions, not just in the first. However, the two-dimensional and three-dimensional FFT routines store the complex data of the second and third dimensions in their entirety.

While the FFT routines accept any size of \( N_1, N_2 \), and \( N_3 \), FFTs can be computed most efficiently when values of \( N_1, N_2 \), and \( N_3 \) can be decomposed into relatively small primes. A real-to-complex or a complex-to-real transform can be computed most efficiently when

\[
N_1, N_2, N_3 = 2^p \times 3^q \times 5^r \times 7^s,
\]

and a complex-to-complex transform can be computed most efficiently when

\[
N_1, N_2, N_3 = 2^p \times 3^q \times 5^r \times 7^s \times 11^t \times 13^u,
\]

where \( p, q, r, s, t, u, \) and \( v \) are integers and \( p, q, r, s, t, u, v \geq 0 \).
The function `xFFTOPT` can be used to determine the optimal sequence length, as shown in CODE EXAMPLE 7-5. Given an input sequence length, the function returns an optimal length that is closest in size to the original length.

**CODE EXAMPLE 7-5  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
```

### 7.2 Cosine and Sine Transforms

Input to the DFT that possess special symmetries occur in various applications. A transform that exploits symmetry usually saves in storage and computational count, such as with the real-to-complex and complex-to-real FFT transforms. The Sun Performance Library cosine and sine transforms are special cases of FFT routines that take advantage of the symmetry properties found in even and odd functions.
Note – Sun Performance Library sine and cosine transform routines are based on the routines contained in FFTPACK (http://www.netlib.org/fftpack/). Routines with a V prefix are vectorized routines that are based on the routines contained in VFFTPACK (http://www.netlib.org/vfftpack/).

### 7.2.1 Fast Cosine and Sine Transform Routines

TABLE 7-5 lists the Sun Performance Library fast cosine and sine transforms. Names of double precision routines are in square brackets. Routines whose name begins with 'V' can compute the transform of one or more sequences simultaneously. Those whose name ends with 'I' are initialization routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fast Cosine Transforms for Even Sequences</strong></td>
<td></td>
</tr>
<tr>
<td>COST [DCOST]</td>
<td>(LEN+1, X, WORK)</td>
</tr>
<tr>
<td>COSTI [DCOSTI]</td>
<td>(LEN+1, WORK)</td>
</tr>
<tr>
<td>VCOST [VDCOST]</td>
<td>(M, LEN+1, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VCOSTI [VDCOSTI]</td>
<td>(LEN+1, TABLE)</td>
</tr>
<tr>
<td><strong>Fast Cosine Transforms for Quarter-Wave Even Sequences</strong></td>
<td></td>
</tr>
<tr>
<td>COSQF [DCOSQF]</td>
<td>(LEN, X, WORK)</td>
</tr>
<tr>
<td>COSQB [DCOSQB]</td>
<td>(LEN, X, WORK)</td>
</tr>
<tr>
<td>COSQI [DCOSQI]</td>
<td>(LEN, WORK)</td>
</tr>
<tr>
<td>VCOSQF [VDCOSQF]</td>
<td>(M, LEN, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VCOSQB [VDCOSQB]</td>
<td>(M, LEN, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VCOSQI [VDCOSQI]</td>
<td>(LEN, TABLE)</td>
</tr>
<tr>
<td><strong>Fast Sine Transforms for Odd Sequences</strong></td>
<td></td>
</tr>
<tr>
<td>SINT [DSINT]</td>
<td>(LEN-1, X, WORK)</td>
</tr>
<tr>
<td>SINTI [DSINTI]</td>
<td>(LEN-1, WORK)</td>
</tr>
<tr>
<td>VSINT [VDSINT]</td>
<td>(M, LEN-1, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VSINTI [VDSINTI]</td>
<td>(LEN-1, TABLE)</td>
</tr>
<tr>
<td><strong>Fast Sine Transforms for Quarter-Wave Odd Sequences</strong></td>
<td></td>
</tr>
<tr>
<td>SINQF [DSINQF]</td>
<td>(LEN, X, WORK)</td>
</tr>
</tbody>
</table>
### Fast Cosine Transforms

A special form of the FFT that operates on real even sequences is the fast cosine transform (FCT). A real sequence $x$ is said to have even symmetry if $x(n) = x(-n)$ where $n = -N + 1, \ldots, 0, \ldots, N$. An FCT of a sequence of length $2N$ requires $N + 1$ input data points and produces a sequence of size $N + 1$. Routine COST computes the FCT of a single real even sequence while VCOST computes the FCT of one or more sequences. Before calling [V]COST, [V]COSTI must be called to compute trigonometric constants and factors associated with input length $N + 1$. The FCT is its own inverse transform. Calling VCOST twice will result in the original $N + 1$ data points. Calling COST twice will result in the original $N + 1$ data points multiplied by $2N$.

An even sequence $x$ with symmetry such that $x(n) = x(-n - 1)$ where $n = -N + 1, \ldots, 0, \ldots, N$ is said to have quarter-wave even symmetry. COSQF and COSQB compute the FCT and its inverse, respectively, of a single real quarter-wave even sequence. VCOSQF and VCOSQB operate on one or more sequences. The results of [V]COSQB are unnormalized, and if scaled by $\frac{1}{N}$, the original sequences are obtained. An FCT of a real sequence of length $2N$ that has quarter-wave even symmetry requires $N$ input data points and produces an $N$-point resulting sequence. Initialization is required before calling the transform routines by calling [V]COSQI.

<p>| TABLE 7-5 Fast Cosine and Sine Transform Routines and Their Arguments |
|---------------------------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINQB [DSINQB]</td>
<td>(LEN, X, WORK)</td>
</tr>
<tr>
<td>SINQI [DSINQI]</td>
<td>(LEN, WORK)</td>
</tr>
<tr>
<td>VSINQF [VDSINQF]</td>
<td>(M, LEN, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VSINQB [VDSINQB]</td>
<td>(M, LEN, X, WORK, LD, TABLE)</td>
</tr>
<tr>
<td>VSINQI [VDSINQI]</td>
<td>(LEN, TABLE)</td>
</tr>
</tbody>
</table>

**Notes:**
- **M**: Number of sequences to be transformed.
- **LEN, LEN-1, LEN+1**: Length of the input sequence or sequences.
- **X**: A real array which contains the sequence or sequences to be transformed. On output, the real transform results are stored in X.
- **TABLE**: Array of constants particular to a transform size that is required by the transform routine. The constants are computed by the initialization routine.
- **WORK**: Workspace required by the transform routine. In routines that operate on a single sequence, WORK also contains constants computed by the initialization routine.
7.2.3 Fast Sine Transforms

Another type of symmetry that is commonly encountered is the odd symmetry where \( x(n) = -x(-n) \) for \( n = -N+1, \ldots, 0, \ldots, N \). As in the case of the fast cosine transform, the fast sine transform (FST) takes advantage of the odd symmetry to save memory and computation. For a real odd sequence \( x \), symmetry implies that \( x(0) = -x(0) = 0 \). Therefore, if \( x \) is of length \( 2N \) then only \( N + 1 \) values of \( x \) are required to compute the FST. Routine \( \text{SINT} \) computes the FST of a single real odd sequence while \( \text{VSINT} \) computes the FST of one or more sequences. Before calling \( \text{[V]SINT} \), \( \text{[V]SINTI} \) must be called to compute trigonometric constants and factors associated with input length \( N - 1 \). The FST is its own inverse transform. Calling \( \text{VSINT} \) twice will result in the original \( N - 1 \) data points. Calling \( \text{SINT} \) twice will result in the original \( N - 1 \) data points multiplied by \( 2^N \).

An odd sequence with symmetry such that \( x(n) = -x(-n - 1) \), where \( n = -N + 1, \ldots, 0, \ldots, N \) is said to have quarter-wave odd symmetry. \( \text{SINQF} \) and \( \text{SINQB} \) compute the FST and its inverse, respectively, of a single real quarter-wave odd sequence while \( \text{VSINQF} \) and \( \text{VSINQB} \) operate on one or more sequences. \( \text{SINQB} \) is unnormalized, so using the results of \( \text{SINQF} \) as input in \( \text{SINQB} \) produces the original sequence scaled by a factor of \( 4N \). However, \( \text{VSINQB} \) is normalized, so a call to \( \text{VSINQF} \) followed by a call to \( \text{VSINQB} \) will produce the original sequence. An FST of a real sequence of length \( 2N \) that has quarter-wave odd symmetry requires \( N \) input data points and produces an \( N \)-point resulting sequence. Initialization is required before calling the transform routines by calling \( \text{[V]SINQI} \).

7.2.4 Discrete Fast Cosine and Sine Transforms and Their Inverse

Sun Performance Library routines use the equations in the following sections to compute the fast cosine and sine transforms and inverse transforms.

7.2.4.1 \( \text{[D]COST} \): Forward and Inverse Fast Cosine Transform (FCT) of a Sequence

The forward and inverse FCT of a sequence is computed as

\[
X(k) = x(0) + 2 \sum_{n=1}^{N-1} x(n) \cos\left(\frac{\pi nk}{N}\right) + x(N) \cos(\pi k), \quad k = 0, \ldots, N.
\]

\( \text{[D]COST} \) Notes:
- \( N + 1 \) values are needed to compute the FCT of an \( N \)-point sequence.
7.2.4.2 \( V[D]COST \): Forward and Inverse Fast Cosine Transforms of Multiple Sequences (VFCT)

The forward and inverse FCTs of multiple sequences are computed as

For \( i = 0, M - 1 \)

\[
X(i, k) = \frac{x(i, 0)}{2N} + \frac{1}{N} \sum_{n=1}^{N-1} x(i, n) \cos\left(\frac{\pi nk}{N}\right) + \frac{x(i, N)}{2N} \cos(\pi k), \quad k = 0, ..., N.
\]

**VFCT Notes**

- \( M \times (N+1) \) values are needed to compute the VFCT of \( M N \)-point sequences.
- The input and output sequences are stored row-wise.
- \( V[D]COST \) is normalized and is its own inverse. When \( V[D]COST \) is called twice, the result will be the original data.

7.2.4.3 \( D[COSQF] \): Forward FCT of a Quarter-Wave Even Sequence

The forward FCT of a quarter-wave even sequence is computed as

\[
X(k) = x(0) + 2 \sum_{n=1}^{N-1} x(n) \cos\left(\frac{\pi n(2k+1)}{2N}\right), \quad k = 0, ..., N-1.
\]

\( N \) values are needed to compute the forward FCT of an \( N \)-point quarter-wave even sequence.

7.2.4.4 \( D[COSQB] \): Inverse FCT of a Quarter-Wave Even Sequence

The inverse FCT of a quarter-wave even sequence is computed as

\[
x(n) = \sum_{k=0}^{N-1} X(k) \cos\left(\frac{\pi n(2k+1)}{2N}\right), \quad n = 0, ..., N-1.
\]

Calling the forward and inverse routines will result in the original input scaled by \( \frac{1}{2N} \).
7.2.4.5 \textbf{V[D]COSQF}: Forward FCT of One or More Quarter-Wave Even Sequences

The forward FCT of one or more quarter-wave even sequences is computed as

For \( i = 0, M - 1 \)

\[
X(i, k) = \frac{1}{N} \left[ x(i, 0) + 2 \sum_{n=1}^{N-1} x(i, n) \cos \left( \frac{\pi n (2k+1)}{2N} \right) \right], \quad k = 0, \ldots, N - 1.
\]

\textbf{V[D]COSQF Notes:}

- The input and output sequences are stored row-wise.
- The transform is normalized so that if the inverse routine \textbf{V[D]COSQB} is called immediately after calling \textbf{V[D]COSQF}, the original data is obtained.

7.2.4.6 \textbf{V[D]COSQB}: Inverse FCT of One or More Quarter-Wave Even Sequences

The inverse FCT of one or more quarter-wave even sequences is computed as

For \( i = 0, M - 1 \)

\[
x(i, n) = \sum_{k=0}^{N-1} X(i, k) \cos \left( \frac{\pi n (2k+1)}{2N} \right), \quad n = 0, \ldots, N - 1.
\]

\textbf{V[D]COSQB Notes:}

- The input and output sequences are stored row-wise.
- The transform is normalized so that if \textbf{V[D]COSQB} is called immediately after calling \textbf{V[D]COSQF}, the original data is obtained.

7.2.4.7 \textbf{[D]SINT}: Forward and Inverse Fast Sine Transform (FST) of a Sequence

The forward and inverse FST of a sequence is computed as

\[
X(k) = 2 \sum_{n=0}^{N-2} x(n) \sin \left( \frac{\pi (n + 1)(k + 1)}{N} \right), \quad k = 0, \ldots, N - 2.
\]

\textbf{[D]SINT Notes:}

- \( N-1 \) values are needed to compute the FST of an \( N \)-point sequence.
7.2.4.8 **V[D]SINT**: Forward and Inverse Fast Sine Transforms of Multiple Sequences (VFST)

The forward and inverse fast sine transforms of multiple sequences are computed as

For \( i = 0, M - 1 \)

\[
X(i, k) = \frac{2}{\sqrt{2N}} \sum_{n=0}^{N-2} x(i, n) \sin \left( \frac{\pi(n+1)(k+1)}{N} \right), \quad k = 0, \ldots, N-2.
\]

**V[D]SINT Notes:**
- \( M \times (N - 1) \) values are needed to compute the VFST of \( M \) \( N \)-point sequences.
- The input and output sequences are stored row-wise.
- **V[D]SINT** is normalized and is its own inverse. Calling **V[D]SINT** twice yields the original data.

7.2.4.9 **[D]SINQF**: Forward FST of a Quarter-Wave Odd Sequence

The forward FST of a quarter-wave odd sequence is computed as

\[
X(k) = 2 \sum_{n=0}^{N-2} x(n) \sin \left( \frac{\pi(n+1)(2k+1)}{2N} \right) + x(N-1) \cos(\pi k), \quad k = 0, \ldots, N-1.
\]

\( N \) values are needed to compute the forward FST of an \( N \)-point quarter-wave odd sequence.

7.2.4.10 **[D]SINQB**: Inverse FST of a Quarter-Wave Odd Sequence

The inverse FST of a quarter-wave odd sequence is computed as

\[
x(n) = 2 \sum_{k=0}^{N-1} X(k) \sin \left( \frac{\pi(n+1)(2k+1)}{2N} \right), \quad n = 0, \ldots, N-1.
\]

Calling the forward and inverse routines will result in the original input scaled by \( \frac{1}{2N} \).
7.2.4.11 \texttt{V[D]SINQF}: Forward FST of One or More Quarter-Wave Odd Sequences

The forward FST of one or more quarter-wave odd sequences is computed as

For \( i = 0, M - 1 \)

\[
X(i, k) = \frac{1}{\sqrt{4N}} \left[ \sum_{n=0}^{N-2} x(n, i) \sin \left( \frac{\pi(n+1)(2k+1)}{2N} \right) + x(N-1, i) \cos \pi k \right], \quad k = 0, \ldots, N-1.
\]

\texttt{V[D]SINQF} Notes:

- The input and output sequences are stored row-wise.
- The transform is normalized so that if the inverse routine \texttt{V[D]SINQB} is called immediately after calling \texttt{V[D]SINQF}, the original data is obtained.

7.2.4.12 \texttt{V[D]SINQB}: Inverse FST of One or More Quarter-Wave Odd Sequences

The inverse FST of one or more quarter-wave odd sequences is computed as

For \( i = 0, M - 1 \)

\[
x(n, i) = \frac{4}{\sqrt{4N}} \sum_{k=0}^{N-1} X(k, i) \sin \left( \frac{\pi(n+1)(2k+1)}{2N} \right), \quad n = 0, \ldots, N-1.
\]

\texttt{V[D]SINQB} Notes:

- The input and output sequences are stored row-wise.
- The transform is normalized, so that if \texttt{V[D]SINQB} is called immediately after calling \texttt{V[D]SINQF}, the original data is obtained.
7.2.5 Fast Cosine Transform Examples

CODE EXAMPLE 7-6 calls COST to compute the FCT and the inverse transform of a real even sequence. If the real sequence is of length $2N$, only $N + 1$ input data points need to be stored and the number of resulting data points is also $N + 1$. The results are stored in the input array.

**CODE EXAMPLE 7-6** Compute FCT and Inverse FCT of Single Real Even Sequence

```fortran
program Drive cost
  implicit none
  integer, parameter :: len=4
  real x(0:len), work(3*(len+1)+15), z(0:len), scale
  integer i
  scale = 1.0/(2.0*len)
call RANDOM_NUMBER(x(0:len))
z(0:len) = x(0:len)
write(*,'(a25,i1,a10,i1,a12)')'Input sequence of length $',
  $', len,' requires ', len+1,' data points'
write(*,'(5(f8.3,2x),/)')(x(i),i=0,len)
call costi(len+1, work)
call cost(len+1, z, work)
write(*,*)'Forward fast cosine transform'
write(*,'(5(f8.3,2x),/)')(z(i),i=0,len)
call cost(len+1, z, work)
write(*,*)
write(*,*)'$ \text{Inverse fast cosine transform (results scaled by } 1/2*N)$'
write(*,'(5(f8.3,2x),/)')(z(i)*scale,i=0,len)
end
```

```
my_system% cat cost.f
my_system% cat Drive cost
```

my_system% f95 -align cost.f -xlic_lib=sunperf
my_system% a.out

Input sequence of length 4 requires 5 data points
0.557 0.603 0.210 0.352 0.867
Forward fast cosine transform
3.753 0.046 1.004 -0.666 -0.066
Inverse fast cosine transform (results scaled by $1/2*N$)
0.557 0.603 0.210 0.352 0.867
CODE EXAMPLE 7-7 calls VCOSQF and VCOSQB to compute the FCT and the inverse FCT, respectively, of two real quarter-wave even sequences. If the real sequences are of length 2N, only N input data points need to be stored, and the number of resulting data points is also N. The results are stored in the input array.

CODE EXAMPLE 7-7  Compute the FCT and the Inverse FCT of Two Real Quarter-wave Even Sequences

```fortran
my_system% cat vcosq.f
program vcosq
    implicit none
    integer,parameter :: len=4, m = 2, ld = m+1
    real x(ld,len),xt(ld,len),work(3*len+15), z(ld,len)
    integer i, j
    call RANDOM_NUMBER(x)
    z = x
    write(*,'(a27,i1)')' Input sequences of length ',len
    do j = 1,m
        write(*,'(a3,i1,a4,4(f5.3,2x),a1,/)
        $     'seq',j,' = (',(x(j,i),i=1,len),')
        end do
    call vcosqi(len, work)
    call vcosqf(m,len, z, xt, ld, work)
    write(*,*)
    $ 'Forward fast cosine transform for quarter-wave even sequences'
    do j = 1,m
        write(*,'(a3,i1,a4,4(f5.3,2x),a1,/)
        $     'seq',j,' = (',(z(j,i),i=1,len),')
        end do
    call vcosqb(m,len, z, xt, ld, work)
    write(*,*)
    $ 'Inverse fast cosine transform for quarter-wave even sequences'
    write(*,*)'(results are normalized)'
    do j = 1,m
        write(*,'(a3,i1,a4,4(f5.3,2x),a1,/)
        $     'seq',j,' = (',(z(j,i),i=1,len),')
        end do
end
```
7.2.6 Fast Sine Transform Examples

In CODE EXAMPLE 7-8, SINT is called to compute the FST and the inverse transform of a real odd sequence. If the real sequence is of length \(2N\), only \(N-1\) input data points need to be stored and the number of resulting data points is also \(N-1\). The results are stored in the input array.

**CODE EXAMPLE 7-8** Compute FST and the Inverse FST of a Real Odd Sequence

```f95
my_system% f95 -dalign vcosq.f -xlic_lib=sunperf
my_system% a.out
Input sequences of length 4
seq1 = (0.557 0.352 0.990 0.539 )
seq2 = (0.603 0.867 0.417 0.156 )
Forward fast cosine transform for quarter-wave even sequences
seq1 = (0.755 -.392 -.029 0.224 )
seq2 = (0.729 0.097 -.091 -.132 )
Inverse fast cosine transform for quarter-wave even sequences
(results are normalized)
seq1 = (0.557 0.352 0.990 0.539 )
seq2 = (0.603 0.867 0.417 0.156 )
```

**CODE EXAMPLE 7-7** Compute the FCT and the Inverse FCT of Two Real Quarter-wave Even Sequences

```f95
my_system% cat sint.f
program Drive sint
implicit none
integer,parameter :: len=4
real x(0:len-2),work(3*(len-1)+15), z(0:len-2), scale
integer i
call RANDOM_NUMBER(x(0:len-2))
z(0:len-2) = x(0:len-2)
scale = 1.0/(2.0*len)
write(*,'(a25,i1,a10,i1,a12)')'Input sequence of length ',', len,' requires ',', len-1,' data points'
write(*,'(3(f8.3,2x),/)')(x(i),i=0,len-2)
call sinti(len-1, work)
call sint(len-1, z, work)
write(*,*)'Forward fast sine transform'
write(*,'(3(f8.3,2x),/)')(z(i),i=0,len-2)
```
In CODE EXAMPLE 7-9 VSINQF and VSINQB are called to compute the FST and inverse FST, respectively, of two real quarter-wave odd sequences. If the real sequence is of length $2N$, only $N$ input data points need to be stored and the number of resulting data points is also $N$. The results are stored in the input array.

CODE EXAMPLE 7-9  Compute FST and Inverse FST of Two Real Quarter-Wave Odd Sequences

```fortran
program vsinq
  implicit none
  integer, parameter :: len=4, m = 2, ld = m+1
  real x(ld,len), xt(ld,len), work(3*len+15), z(ld,len)
  integer i, j
  call RANDOM_NUMBER(x)
  z = x
  write(*,'(a27,i1)')' Input sequences of length ',len
  do j = 1,m
    write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'seq',j,' = (',(x(j,i),i=1,len),')'
  end do
  call vsinqi(len, work)
  call vsinqf(m,len, z, xt, ld, work)
  write(*,*)
  write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'Forward fast sine transform for quarter-wave odd sequences'
  do j = 1,m
    write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'seq',j,' = (',(z(j,i),i=1,len),')'
  end do
  end program vsinq
```

In CODE EXAMPLE 7-9 VSINQF and VSINQB are called to compute the FST and inverse FST, respectively, of two real quarter-wave odd sequences. If the real sequence is of length $2N$, only $N$ input data points need to be stored and the number of resulting data points is also $N$. The results are stored in the input array.

CODE EXAMPLE 7-9  Compute FST and Inverse FST of Two Real Quarter-Wave Odd Sequences

```fortran
program vsinq
  implicit none
  integer, parameter :: len=4, m = 2, ld = m+1
  real x(ld,len), xt(ld,len), work(3*len+15), z(ld,len)
  integer i, j
  call RANDOM_NUMBER(x)
  z = x
  write(*,'(a27,i1)')' Input sequences of length ',len
  do j = 1,m
    write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'seq',j,' = (',(x(j,i),i=1,len),')'
  end do
  call vsinqi(len, work)
  call vsinqf(m,len, z, xt, ld, work)
  write(*,*)
  write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'Forward fast sine transform for quarter-wave odd sequences'
  do j = 1,m
    write(*,'(a3,i1,a4,4(f5.3,2x),a1,/))' 'seq',j,' = (',(z(j,i),i=1,len),')'
  end do
  end program vsinq
```
7.3 Convolution and Correlation

Two applications of the FFT that are frequently encountered especially in the signal processing area are the discrete convolution and discrete correlation operations.

7.3.1 Convolution

Given two functions $x(t)$ and $y(t)$, the Fourier transform of the convolution of $x(t)$ and $y(t)$, denoted as $x * y$, is the product of their individual Fourier transforms:

$$\text{DFT}(x * y) = X \odot Y$$

where $*$ denotes the convolution operation and $\odot$ denotes pointwise multiplication.

Typically, $x(t)$ is a continuous and periodic signal that is represented discretely by a set of $N$ data points $x_j$, $j = 0, ..., N-1$, sampled over a finite duration, usually for one period of $x(t)$ at equal intervals. $y(t)$ is usually a response that starts out as zero, peaks to a maximum value, and then returns to zero. Discretizing $y(t)$ at equal
intervals produces a set of \( N \) data points, \( y_k \), \( k = 0, \ldots, N-1 \). If the actual number of samplings in \( y_k \) is less than \( N \), the data can be padded with zeros. The discrete convolution can then be defined as

\[
(x * y)_j = \sum_{k = \frac{-N}{2} + 1}^{N/2} x_{j-k} y_k, \quad j = 0, \ldots, N-1.
\]

The values of \( y_{k} \), \( k = \frac{-N}{2} + 1, \ldots, \frac{N}{2} \), are the same as those of \( k = 0, \ldots, N-1 \) but in the wrap-around order.

The Sun Performance Library routines allow the user to compute the convolution by using the definition above with \( k = 0, \ldots, N-1 \), or by using the FFT. If the FFT is used to compute the convolution of two sequences, the following steps are performed:

- Compute \( X = \) forward FFT of \( x \)
- Compute \( Y = \) forward FFT of \( y \)
- Compute \( Z = X \odot Y \Leftrightarrow \text{DFT}(x \ast y) \)
- Compute \( z = \) inverse FFT of \( Z \)

One interesting characteristic of convolution is that the product of two polynomials is actually a convolution. A product of an \( m \)-term polynomial

\[
a(x) = a_0 + a_1 x + \ldots + a_{m-1} x^{m-1}
\]

and an \( n \)-term polynomial

\[
b(x) = b_0 + b_1 x + \ldots + b_{n-1} x^{n-1}
\]

has \( m + n - 1 \) coefficients that can be obtained by

\[
c_k = \sum_{j = \max(k - (m-1), 0)}^{\min(k, n-1)} a_j b_{k-j},
\]

where \( k = 0, \ldots, m + n - 2 \).

### 7.3.2 Correlation

Closely related to convolution is the correlation operation. It computes the correlation of two sequences directly superposed or when one is shifted relative to the other. As with convolution, we can compute the correlation of two sequences efficiently as follows using the FFT:

- Compute the FFT of the two input sequences.
- Compute the pointwise product of the resulting transform of one sequence and the complex conjugate of the transform of the other sequence.
- Compute the inverse FFT of the product.
The routines in the Performance Library also allow correlation to be computed by the following definition:

\[
\text{Corr}(x, y)_j = \sum_{k=0}^{N-1} x_{j+k} y_k, \quad j = 0, \ldots, N-1.
\]

There are various ways to interpret the sampled input data of the convolution and correlation operations. The argument list of the convolution and correlation routines contain parameters to handle cases in which

- The signal and/or response function can start at different sampling time
- The user might want only part of the signal to contribute to the output
- The signal and/or response function can begin with one or more zeros that are not explicitly stored.

### 7.3.3 Sun Performance Library Convolution and Correlation Routines

Sun Performance Library contains the convolution routines shown in **TABLE 7-6**.

**TABLE 7-6** Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCNVCOR</td>
<td>CNVCOR, FOUR, NX, X, IFX, INCX, NY, NPRE, M, Y, IFY, INC1Y, INC2Y, NZ, K, Z, IFZ, INC1Z, INC2Z, WORK, LWORK</td>
<td>Convolution or correlation of a filter with one or more vectors</td>
</tr>
<tr>
<td>DCNVCOR</td>
<td>CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY, MX, NX, X, LDX, MY, NY, NPRE, NPRE, Y, LDY, MZ, NZ, Z, LDZ, WORKIN, LWORK</td>
<td>Two-dimensional convolution or correlation of two matrices</td>
</tr>
<tr>
<td>SCNVCOR2</td>
<td>CNVCOR, METHOD, TRANSX, SCRATCHX, TRANSY, SCRATCHY, MX, NX, X, LDX, MY, NY, NPRE, NPRE, Y, LDY, MZ, NZ, Z, LDZ, WORKIN, LWORK</td>
<td>Two-dimensional convolution or correlation of two matrices</td>
</tr>
<tr>
<td>ZCNVCOR</td>
<td>IFZ, INC1Z, INC2Z, WORK, LWORK</td>
<td>Two-dimensional convolution or correlation of two matrices</td>
</tr>
<tr>
<td>SWIENER</td>
<td>N_POINTS, ACOR, XCOR, FLTR, EROP, ISW, IERR</td>
<td>Wiener deconvolution of two signals</td>
</tr>
<tr>
<td>DWIENER</td>
<td>FLTR, EROP, ISW, IERR</td>
<td>Wiener deconvolution of two signals</td>
</tr>
</tbody>
</table>

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.
### 7.3.4 Arguments for Convolution and Correlation Routines

The one-dimensional convolution and correlation routines use the arguments shown in Table 7-7.

**Table 7-7: Arguments for One-Dimensional Convolution and Correlation Routines**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNVCOR</td>
<td>'V' or 'v' specifies that convolution is computed. 'R' or 'r' specifies that correlation is computed.</td>
</tr>
<tr>
<td>FOUR</td>
<td>'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.*</td>
</tr>
<tr>
<td>NX</td>
<td>Length of filter vector, where NX ≥ 0.</td>
</tr>
<tr>
<td>X</td>
<td>Filter vector</td>
</tr>
<tr>
<td>IFX</td>
<td>Index of first element of X, where NX ≥ IFX ≥ 1</td>
</tr>
<tr>
<td>INCX</td>
<td>Stride between elements of the vector in X, where INCX &gt; 0.</td>
</tr>
<tr>
<td>NY</td>
<td>Length of input vectors, where NY ≥ 0.</td>
</tr>
<tr>
<td>NPRE</td>
<td>Number of implicit zeros prefixed to the Y vectors, where NPRE ≥ 0.</td>
</tr>
<tr>
<td>M</td>
<td>Number of input vectors, where M ≥ 0.</td>
</tr>
<tr>
<td>Y</td>
<td>Input vectors.</td>
</tr>
<tr>
<td>IFY</td>
<td>Index of the first element of Y, where NY ≥ IFY ≥ 1</td>
</tr>
<tr>
<td>INC1Y</td>
<td>Stride between elements of the input vectors in Y, where INC1Y &gt; 0.</td>
</tr>
<tr>
<td>INC2Y</td>
<td>Stride between input vectors in Y, where INC2Y &gt; 0.</td>
</tr>
<tr>
<td>NZ</td>
<td>Length of the output vectors, where NZ ≥ 0.</td>
</tr>
<tr>
<td>K</td>
<td>Number of Z vectors, where K ≥ 0. If K &lt; M, only the first K vectors will be processed. If K &gt; M, all input vectors will be processed and the last M-K output vectors will be set to zero on exit.</td>
</tr>
<tr>
<td>Z</td>
<td>Result vectors</td>
</tr>
<tr>
<td>IFZ</td>
<td>Index of the first element of Z, where NZ ≥ IFZ ≥ 1</td>
</tr>
<tr>
<td>INC1Z</td>
<td>Stride between elements of the output vectors in Z, where INCYZ &gt; 0.</td>
</tr>
</tbody>
</table>
The two-dimensional convolution and correlation routines use the arguments shown in TABLE 7-8.

### TABLE 7-8 Arguments for Two-Dimensional Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC2Z</td>
<td>Stride between output vectors in Z, where INC2Z &gt; 0.</td>
</tr>
<tr>
<td>WORK</td>
<td>Work array</td>
</tr>
<tr>
<td>LWORK</td>
<td>Length of work array</td>
</tr>
</tbody>
</table>

* 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.

### TABLE 7-7 Arguments for One-Dimensional Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNVCOR</td>
<td>‘V’ or ‘v’ specifies that convolution is computed.</td>
</tr>
<tr>
<td>DCNVCOR</td>
<td>‘D’ or ‘d’ specifies that the direct method is used, where the convolution or correlation is computed from the definition of convolution and correlation.</td>
</tr>
<tr>
<td>CCNVCOR</td>
<td>‘C’ or ‘c’ specifies that correlation is computed.</td>
</tr>
<tr>
<td>SCNVCOR</td>
<td>‘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.</td>
</tr>
<tr>
<td>TRANSX</td>
<td>‘N’ or ‘n’ specifies that X is the filter matrix</td>
</tr>
<tr>
<td>TRANSY</td>
<td>‘N’ or ‘n’ specifies that Y is the input matrix</td>
</tr>
<tr>
<td>SCRATCHX</td>
<td>‘N’ or ‘n’ specifies that X must be preserved</td>
</tr>
<tr>
<td>SCRATCHY</td>
<td>‘N’ or ‘n’ specifies that Y must be preserved</td>
</tr>
<tr>
<td>MX</td>
<td>Number of rows in the filter matrix X, where MX ≥ 0</td>
</tr>
<tr>
<td>NX</td>
<td>Number of columns in the filter matrix X, where NX ≥ 0</td>
</tr>
<tr>
<td>X</td>
<td>Filter matrix. X is unchanged on exit when SCRATCHX is ‘N’ or ‘n’ and undefined on exit when SCRATCHX is ‘S’ or ‘s’.</td>
</tr>
<tr>
<td>LDX</td>
<td>Leading dimension of array containing the filter matrix X.</td>
</tr>
<tr>
<td>MY</td>
<td>Number of rows in the input matrix Y, where MY ≥ 0.</td>
</tr>
</tbody>
</table>
7.3.5 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 7-11. 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 7-9.

**TABLE 7-9** Arguments Affecting Minimum Work Array Size for Two-Dimensional Routines: SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MX</td>
<td>Number of rows in the filter matrix</td>
</tr>
<tr>
<td>MY</td>
<td>Number of rows in the input matrix</td>
</tr>
<tr>
<td>MZ</td>
<td>Number of output vectors</td>
</tr>
<tr>
<td>NX</td>
<td>Number of columns in the filter matrix</td>
</tr>
<tr>
<td>NY</td>
<td>Number of columns in the input matrix</td>
</tr>
<tr>
<td>NZ</td>
<td>Length of output vectors</td>
</tr>
<tr>
<td>MPRE</td>
<td>Number of implicit zeros prefixed to each row of the input matrix</td>
</tr>
<tr>
<td>NPRE</td>
<td>Number of implicit zeros prefixed to each column of the input matrix</td>
</tr>
<tr>
<td>MPOST</td>
<td>MAX(0, MZ-MYC)</td>
</tr>
<tr>
<td>NPOST</td>
<td>MAX(0, NZ-NYC)</td>
</tr>
<tr>
<td>MYC</td>
<td>MPRE + MPOST + MYC_INIT, where MYC_INIT depends upon filter and input matrices, as shown in TABLE 7-10</td>
</tr>
<tr>
<td>NYC</td>
<td>NPRE + NPOST + NYC_INIT, where NYC_INIT depends upon filter and input matrices, as shown in TABLE 7-10</td>
</tr>
</tbody>
</table>

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

**TABLE 7-10** MYC_INIT and NYC_INIT Dependencies

<table>
<thead>
<tr>
<th>Y</th>
<th>Transpose(Y)</th>
<th>X</th>
<th>Transpose(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MYC_INIT</td>
<td>MAX(MX, MY)</td>
<td>MAX(NX, MY)</td>
<td>MAX(MX, NY)</td>
</tr>
<tr>
<td>NYC_INIT</td>
<td>MAX(NX, NY)</td>
<td>MAX(MX, NY)</td>
<td>MAX(NX, MY)</td>
</tr>
</tbody>
</table>
The values assigned to the minimum work array size is shown in TABLE 7-11.

**TABLE 7-11** Minimum Dimensions and Data Types for WORK Work Array Used With Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Minimum Work Array Size (WORK)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCNVCOR, DCNVCOR</td>
<td>4*(MAX(NX,NPRE+NY) + MAX(0,NZ-NY))</td>
<td>REAL, REAL*8</td>
</tr>
<tr>
<td>CCHVCOR, ZCNVCOR</td>
<td>2*(MAX(NX,NPRE+NY) + MAX(0,NZ-NY))</td>
<td>COMPLEX, COMPLEX*16</td>
</tr>
<tr>
<td>SCNVCOR2, DCNVCOR2</td>
<td>MY + NY + 30</td>
<td>COMPLEX, COMPLEX*16</td>
</tr>
<tr>
<td>CCNVCOR2, ZCNVCOR2</td>
<td>If MY = NY: MYC + 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If MY ≠ NY: MYC + NYC + 16</td>
<td>COMPLEX, COMPLEX*16</td>
</tr>
</tbody>
</table>

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

### 7.3.6 Sample Program: Convolution

**CODE EXAMPLE 7-10** uses CCNVCOR to perform FFT convolution of two complex vectors.

**CODE EXAMPLE 7-10** One-Dimensional Convolution Using Fourier Transform Method and COMPLEX Data

```fortran
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
```
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, \( \text{INCX} = 1 \), \( \text{INC1Y} = 1 \), \( \text{INC2Y} \geq NY \), \( \text{INC1Z} = 1 \), \( \text{INC2Z} \geq NZ \). 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 \( \text{INCX} = 1 \), \( \text{INC1Y} \geq NY \), \( \text{INC2Y} = 1 \), \( \text{INC1Z} \geq NZ \), and \( \text{INC2Z} = 1 \).
Convolution can be used to compute the products of polynomials. CODE EXAMPLE 7-11 uses SCNVCOR to compute the product of $1 + 2x + 3x^2$ and $4 + 5x + 6x^2$.

CODE EXAMPLE 7-11  One-Dimensional Convolution Using Fourier Transform Method and REAL Data

```fortran
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 7-12 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 7-12** Convolution Used to Compute the Product of a Vector and Circulant Matrix

```fortran
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 7-13  Two-Dimensional Convolution Using Direct Method

```c
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
```

```
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
```
7.4 References

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


8

Interval BLAS Routines

8.1 Introduction

This chapter provides a brief overview of an interval Fortran 95 version of the basic linear algebra subroutine (BLAS) library. The interval BLAS version is referred to as the IBLAS library. For a more complete description of the IBLAS library routines, see the white paper *Interval Version of the Basic Linear Algebra Subprograms (IBLAS)*.

For information on the Fortran 95 interfaces and types of arguments used in each IBLAS routine, see the section 3P man pages for the individual routines. For example, to display the man page for the `SFFTFC` routine, type `man -s 3P sfftfc`. Routine names must be lowercase.


**Note** — For the Sun Studio Fortran 95 IBLAS routines, information contained in the *Interval Version of the Basic Linear Algebra Subprograms (IBLAS)* white paper supersedes interval information contained in the *Basic Linear Algebra Subprogram Technical (BLAST) Forum Standard* document that is available from NetLib.

8.1.1 Intervals

Intervals have a dual identity as intervals of *numbers* and as *sets* of numbers. The empty interval $\emptyset$ contains no members and is the same as the empty set in the theory of sets. In computer input and output, the empty interval is denoted $\langle \rangle$. 
For more information on intrinsic Fortran 95 compiler support for interval data types, see the *Fortran 95 Interval Arithmetic Programming Reference* and the interval white papers referenced therein.

## 8.2 IBLAS Routine Names

This section summarizes IBLAS naming conventions derived from the BLAS specification. “Language Bindings” on page 128 contains a list of IBLAS routine names organized into the following groups. For the corresponding detailed Fortran language bindings, see the IBLAS man pages or the IBLAS white paper.

As in the BLAS, mathematical operations and routines are grouped into:

- Vector Operations Tables, listed in TABLE 8-2 through TABLE 8-4.
- Matrix-Vector Operations Table, listed in TABLE 8-5.
- Matrix Operations Tables, listed in TABLE 8-6 through TABLE 8-8.

New interval-specific routines are grouped into:

- Set Operations on Vectors, listed in TABLE 8-9.
- Set Operations on Matrices, listed in TABLE 8-10.
- Utility Functions of Vectors, listed in TABLE 8-11.
- Utility Functions of Matrices, listed in TABLE 8-12.

### 8.2.1 Naming Conventions

Except that the suffix \_I or \_i is added, IBLAS routines are named the same as the corresponding BLAS routines described in (ref BLAST Standard). IBLAS routine names have the same prefixes as the BLAS routines. Routines with prefixes identify the matrix type. TABLE 8-1 lists the IBLAS prefixes and matrix types.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Matrix Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GE</td>
<td>General</td>
</tr>
<tr>
<td>GB</td>
<td>General Banded</td>
</tr>
<tr>
<td>SY</td>
<td>Symmetric</td>
</tr>
<tr>
<td>SB</td>
<td>Symmetric Banded</td>
</tr>
<tr>
<td>SP</td>
<td>Symmetric Packed</td>
</tr>
</tbody>
</table>

---

[empty]
As in the BLAS, sparse or complex interval matrices are not treated.

A number of interval-specific, set, and utility IBLAS routines are given new BLAS-style names. See TABLE 8-9 through TABLE 8-12.

8.3 Fortran Interface

The IBLAS Fortran bindings are implemented in a module. Its interface block defines the default interval data type to be \texttt{TYPE(INTERVAL)}.

Interval BLAS routines are consistent with regard to:

- Generic interfaces
- Precision
- Rank
- Assumed-shape arrays
- Derived types
- Operator arguments.

Error handling is described in the \textit{Basic Linear Algebra Subprogram Technical (BLAST) Forum Standard} and in the IBLAS white paper.

Numeric error handling is not required because exceptions are not possible in the closed interval system implemented in the Sun Studio \texttt{f95} compiler. Argument inconsistency errors are handled as described in IBLAS white paper, the IBLAS man pages, and the BLAST standard.

In general, actual argument shape inconsistencies cause IBLAS routines to return the largest impossible value of -1 for integer indices, a default NaN for \texttt{REAL} values, and the interval $\Re^* = [-\infty, +\infty]$ for computed intervals. The normal BLAS error handling mechanism is also used to communicate actual-parameter errors.
8.3.1 Binding Format

Each interface is summarized as a SUBROUTINE or FUNCTION statement, in which all the required and optional arguments appear. Optional arguments are grouped in square brackets after the required arguments. Binding format is illustrated with the Scaled Vector Sum Update (AXPY_I) routine.

```fortran
SUBROUTINE axpby_i( x, y [, alpha] [, beta] )
    TYPE(INTERVAL) (<wp>), INTENT (IN) :: x (:)
    TYPE(INTERVAL) (<wp>), INTENT (INOUT) :: y (:)
    TYPE(INTERVAL) (<wp>), INTENT (IN), OPTIONAL :: alpha, beta
```

Because generic interfaces are used, the working precision, denoted <wp> is implicitly defined by the following actual arguments:

```fortran
<wp> ::= KIND(4) | KIND(8) | KIND(16)
```

Variables in IBLAS routines are INTEGER, REAL, or TYPE(INTERVAL). See the IBLAS man pages or the IBLAS white pager for individual routine bindings.

8.3.2 Language Bindings

This section is a brief overview of the IBLAS Fortran routine names and their function. With the one exception of the CANCEL routines, which perform the same operation as the .DSUB. operator in f95, vector and set reductions and operations are the same as in the BLAS. The CANCEL routines and all the vector and matrix set operations and utilities are interval-specific. For interval-specific routines, the f95 equivalent scalar routines are also shown in TABLE 8-3 and TABLE 8-9 through TABLE 8-12. For clarity, lowercase and uppercase Fortran variable names are used to distinguish point from interval types. See TABLE A-12 for an alphabetical list of all the IBLAS routines.

**TABLE 8-2** Vector Reductions

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOT_I</td>
<td>Dot Product</td>
</tr>
<tr>
<td>NORM_I</td>
<td>Vector Norms</td>
</tr>
<tr>
<td>SUM_I</td>
<td>Sum</td>
</tr>
<tr>
<td>AMIN_VAL_I</td>
<td>Minimum Absolute Value and Location</td>
</tr>
<tr>
<td>AMAX_VAL_I</td>
<td>Maximum Absolute Value and Location</td>
</tr>
<tr>
<td>SUMSQ_I</td>
<td>Scaled Sum of Squares and Update</td>
</tr>
</tbody>
</table>
### TABLE 8-3  Add or Cancel Vectors

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>$f95$ Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSCALE_I</td>
<td>Reciprocally Scale Vector</td>
<td></td>
</tr>
<tr>
<td>AXPBY_I</td>
<td>Add Scaled Vectors and Update</td>
<td></td>
</tr>
<tr>
<td>WAXPBY_I</td>
<td>Add Scaled Vectors</td>
<td></td>
</tr>
<tr>
<td>CANCEL_I</td>
<td>Cancel Scaled Vectors and Update</td>
<td>$Y = a<em>X.DSUB. b</em>Y$</td>
</tr>
<tr>
<td>WCANCEL_I</td>
<td>Cancel Scaled Vectors</td>
<td>$W = a<em>X.DSUB. b</em>Y$</td>
</tr>
<tr>
<td>SUMSQ_I</td>
<td>Scaled Sum of Squares and Update</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 8-4  Vector Movements

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY_I</td>
<td>Vector Copy</td>
</tr>
<tr>
<td>SWAP_I</td>
<td>Vector Swap</td>
</tr>
<tr>
<td>PERMUTE_I</td>
<td>Permute Vector and Update</td>
</tr>
</tbody>
</table>

### TABLE 8-5  Matrix-Vector Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GE,GB)MV_I</td>
<td>General Matrix-Vector Product and Update</td>
</tr>
<tr>
<td>(SY,SB,SP)MV_I</td>
<td>Symmetric Matrix-Vector Product and Update</td>
</tr>
<tr>
<td>(TR,TB,TP)MV_I</td>
<td>Triangular Matrix-Vector Product and Update</td>
</tr>
<tr>
<td>(TR,TB,TP)SV_I</td>
<td>Triangular Matrix Solve and Update</td>
</tr>
<tr>
<td>GER_I</td>
<td>General-Matrix Rank-One Update</td>
</tr>
<tr>
<td>(SY,SP)R_I</td>
<td>Symmetric-Matrix Rank-One Update</td>
</tr>
</tbody>
</table>
### TABLE 8-6  \(O(n^2)\) Matrix Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[GE, GB, SY, SB, SP, TR, TB, TP]_NORM_I</td>
<td>Matrix Norms</td>
</tr>
<tr>
<td>[GE, GB]_DIAG_SCALE_I</td>
<td>Scale General Matrix Rows or Columns and Update</td>
</tr>
<tr>
<td>[GE, GB]_LRSCALE_I</td>
<td>Scale General Matrix Rows and Columns and Update</td>
</tr>
<tr>
<td>[SY, SB, SP]_LRSCALE_I</td>
<td>Scale Symmetric Matrix Rows and Columns and Update</td>
</tr>
<tr>
<td>[GE, GB, SY, SB, SP, TR, TB, TP]_ACC_I</td>
<td>Add Scaled Matrices and Update</td>
</tr>
<tr>
<td>[GE, GB, SY, SB, SP, TR, TB, TP]_ADD_I</td>
<td>Add Scaled Matrices</td>
</tr>
</tbody>
</table>

### TABLE 8-7  \(O(n^3)\) Matrix Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM_I</td>
<td>General Matrix-Matrix Product and Update</td>
</tr>
<tr>
<td>SYMM_I</td>
<td>Symmetric-General Matrix-Matrix Product and Update</td>
</tr>
<tr>
<td>TRMM_I</td>
<td>Triangular-General Matrix-Matrix Product and Update</td>
</tr>
<tr>
<td>TRSM_I</td>
<td>Triangular Matrix Solve</td>
</tr>
</tbody>
</table>

### TABLE 8-8  Matrix Movements

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{GE, GB, SY, SB, SP, TR, TB, TP}_COPY_I</td>
<td>Copy Matrix</td>
</tr>
<tr>
<td>GE_TRANS_I</td>
<td>Transpose Matrix</td>
</tr>
<tr>
<td>GE_PERMUTE_I</td>
<td>Permute Matrix</td>
</tr>
</tbody>
</table>

### TABLE 8-9  Vector Set Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>\texttt{$95\ Equivalent}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLOSEV_I</td>
<td>Enclose Vector Test</td>
<td>\texttt{X.SB.Y}</td>
</tr>
<tr>
<td>INTERIORV_I</td>
<td>Vector Interior Test</td>
<td>\texttt{X.INT.Y}</td>
</tr>
<tr>
<td>DISJOINTV_I</td>
<td>Disjoint Vector Test</td>
<td>\texttt{X.DJ.Y}</td>
</tr>
</tbody>
</table>
TABLE 8-9  Vector Set Operations (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>f95 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERSECTV_I</td>
<td>Intersect Vectors and Update</td>
<td>Y = X.IX.Y</td>
</tr>
<tr>
<td>WINTERSECTV_I</td>
<td>Intersect Vectors</td>
<td>W = X.IX.Y</td>
</tr>
<tr>
<td>HULLV_I</td>
<td>Hull of Vectors and Update</td>
<td>Y = X.IH.Y</td>
</tr>
<tr>
<td>WHULLV_I</td>
<td>Hull of Vectors</td>
<td>W = X.IH.Y</td>
</tr>
</tbody>
</table>

TABLE 8-10  Matrix Set Operations

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Name</th>
<th>Operation</th>
<th>f95 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLOSHEME_I</td>
<td>Enclose Matrix Test</td>
<td>A.SB.B</td>
<td></td>
</tr>
<tr>
<td>INTERIORM_I</td>
<td>Matrix Interior Test</td>
<td>A.INT.B</td>
<td></td>
</tr>
<tr>
<td>DISJOINTM_I</td>
<td>Disjoint Matrix Test</td>
<td>A.DJ.B</td>
<td></td>
</tr>
<tr>
<td>{GE, GB, SY, SB, SP, TR, TB, TP}</td>
<td>INTERSECTM_I</td>
<td>Intersect Matrices and Update</td>
<td>B = X.IX.B</td>
</tr>
<tr>
<td>WINTERSECTM_I</td>
<td>Intersect Matrices</td>
<td>W = X.IX.B</td>
<td></td>
</tr>
<tr>
<td>HULLM_I</td>
<td>Hull of Matrices and Update</td>
<td>B = X.IH.B</td>
<td></td>
</tr>
<tr>
<td>WHULLM_I</td>
<td>Hull of Matrices</td>
<td>W = X.IH.B</td>
<td></td>
</tr>
</tbody>
</table>

Note: Prefix depends upon matrix type and applies to all routine names in this table.

TABLE 8-11  Vector Utilities

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>f95 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTYV_I</td>
<td>Empty Vector Element Test and Location</td>
<td>ISEMPY(X)</td>
</tr>
<tr>
<td>INFV_I</td>
<td>Vector Infimum</td>
<td>v = INF(X)</td>
</tr>
<tr>
<td>SUPV_I</td>
<td>Vector Supremum</td>
<td>v = SUP(X)</td>
</tr>
<tr>
<td>MIDV_I</td>
<td>Vector Midpoint</td>
<td>v = MID(X)</td>
</tr>
<tr>
<td>WIDTHV_I</td>
<td>Vector Width</td>
<td>v = WID(X)</td>
</tr>
<tr>
<td>INTERVALV_I</td>
<td>Vector Type Conversion to Interval</td>
<td>X = INTERVAL(u,v)</td>
</tr>
</tbody>
</table>
8.3.2.1 Example

The Sun Interval Performance Library has a module file that contains the F95 binding with interfaces for individual IBLAS routines. User codes reference the routines by their generic names. To access the IBLAS routines, a Fortran 95 code must contain the statement

```
USE IASUNPERF
```

A set of frequently-used variables is defined in a second module file. User codes can access these derived types through the USE statement

```
USE IATYPESUNPERF
```

Their values are listed in the following table:

### Table 8-12 Matrix Utilities

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Name</th>
<th>Operation</th>
<th>f95 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTM_I</td>
<td>Empty Matrix Element Test and Location</td>
<td>IEMPTY(A)</td>
<td></td>
</tr>
<tr>
<td>INFM_I</td>
<td>Matrix Infimum</td>
<td></td>
<td>INF(A)</td>
</tr>
<tr>
<td>(GE, GB, SY, SB, SE, TR, TB, TP)_</td>
<td>SUPM_I</td>
<td>Matrix Supremum</td>
<td>SUP(A)</td>
</tr>
<tr>
<td>MIDM_I</td>
<td>Matrix Midpoint</td>
<td>c = MID(A)</td>
<td></td>
</tr>
<tr>
<td>WIDTHM_I</td>
<td>Matrix Width</td>
<td>c = WID(A)</td>
<td></td>
</tr>
<tr>
<td>INTERVALM_I</td>
<td>Matrix Type Conversion to Interval</td>
<td>A = INTERVAL(b,c)</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Prefix depends upon matrix type and applies to all routine names in this table.

### Table 8-13 Interval BLAS User-Defined Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>blas_trans_type</td>
<td>blas_no_trans, blas_trans, blas_conj_trans</td>
</tr>
<tr>
<td>blas_uplo_type</td>
<td>blas_upper, blas_lower</td>
</tr>
<tr>
<td>blas_diag_type</td>
<td>blas_non_unit_diag, blas_unit_diag</td>
</tr>
<tr>
<td>blas_side_type</td>
<td>blas_left_side, blas_right_side</td>
</tr>
<tr>
<td>blas_norm_type</td>
<td>blas_one_norm, blas_real_one_norm, blas_two_norm, blas_frobenius_norm, blas_inf_norm, blas_real_inf_norm, blas_max_norm, blas_real_max_norm</td>
</tr>
</tbody>
</table>
When compiling a program with interval data types, use compiler flag `-xia` to enable interval arithmetic extensions in the Sun F95 compiler. The Sun Interval Performance Library is linked into an application with the `-xlic_lib=suniperf` switch as follows:

```
my_system% f95 -dalign -xia my_file.f -xlic_lib=suniperf
```

Below is an example that shows how to compile, link and execute a program that computes the interval dot product.

**CODE EXAMPLE 8-1  Interval BLAS Example**

```
PROGRAM IBLAS_EXAMPLE
USE IASUNPERF
USE IATYPESUNPERF
IMPLICIT NONE

INTEGER, PARAMETER :: KINDMY=4
INTERVAL(KIND = KINDMY), ALLOCATABLE :: x(:), y(:)
INTERVAL(KIND = KINDMY), ALLOCATABLE :: z(:,:)
INTERVAL(KIND = KINDMY) :: alpha, beta, r, solution
INTEGER N, i
TYPE(blas_norm_type) norm

norm = blas_frobenius_norm
N = 8
ALLOCATE (x(N))
ALLOCATE (y(N))
alpha = [1._KINDMY]
beta = [1._KINDMY]
CALL RANDOM_Number(r)
x = 2._KINDMY
y = 0.5_KINDMY
solution = [0._KINDMY]

DO i = 1,N
    solution = solution + x(i)*y(i)
END DO

solution = beta*r + alpha*solution

CALL DOT_I(x, y, r, alpha, beta)
PRINT*, "calculated dot product: ", r
PRINT*, "expected dot product: ", solution
```
Running the above example:

```fortran
my_system% f95 -xia -dalign -o iblas_example iblas_example.F90 \
             -xlic_lib=suniperf
my_system% ./iblas_example
```

- calculated dot product:  [8.5568342,8.6031237]
- expected dot product:    [8.5568342,8.6031237]
- Frobenius norm:          [8.0,8.0]

8.3.3 References

The following white paper is available online. See the Interval Arithmetic readme for the location of this file.

“Interval Version of the Basic Linear Algebra Subprograms Standard (IBLAS),” derived by G.W. Walster from the draft INTERVAL BLAS Chapter 5 prepared by Chenyi Hu, et. al., to be included in the Basic Linear Algebra Subprogram Technical (BLAST) Forum Standard.
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 `xBDSQR` is available as four routines that operate with the following data types:

- SBDSQR – Single data type
- DBDSQR – 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.

If a routine name is not available for S, D, C, and Z, the `x` prefix will not be used and each routine name will be listed. Also available (but not listed) in 64-bit enable operating environments are the corresponding routines in 64-bit. Their names are denoted by the `_64` suffix. For example, the 64-bit version of `xBDSQR` is:

- SBDSQR_64
- DBDSQR_64
- CBDSQR_64
- ZBDSQR_64.
LAPACK Routines

**TABLE A-1** lists the Sun Performance Library LAPACK routines. (P) denotes routines that are parallelized.

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bidiagonal Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>SBDSDC or DBDSDC</td>
<td>Computes the singular value decomposition (SVD) of a bidirectional matrix, using a divide and conquer method.</td>
</tr>
<tr>
<td>xBDSQR</td>
<td>Computes SVD of real upper or lower bidiagonal matrix, using the bidirectional QR algorithm.</td>
</tr>
<tr>
<td><strong>Diagonal Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>SDISNA or DDISNA</td>
<td>Computes the reciprocal condition numbers for eigenvectors of real symmetric or complex Hermitian matrix.</td>
</tr>
<tr>
<td><strong>General Band Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>xGBBRD</td>
<td>Reduces real or complex general band matrix to upper bidiagonal form.</td>
</tr>
<tr>
<td>xGBCON</td>
<td>Estimates the reciprocal of the condition number of general band matrix using LU factorization.</td>
</tr>
<tr>
<td>xGBEQU</td>
<td>Computes row and column scalings to equilibrate a general band matrix and reduce its condition number.</td>
</tr>
<tr>
<td>xGBRFS</td>
<td>Refines solution to general banded system of linear equations.</td>
</tr>
<tr>
<td>xGBSV</td>
<td>Solves a general banded system of linear equations (simple driver).</td>
</tr>
<tr>
<td>xGBSVX</td>
<td>Solves a general banded system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xGBTRF</td>
<td>LU factorization of a general band matrix using partial pivoting with row interchanges.</td>
</tr>
<tr>
<td>xGBTRS (P)</td>
<td>Solves a general banded system of linear equations, using the factorization computed by xGBTRF.</td>
</tr>
<tr>
<td><strong>General Matrix (Unsymmetric or Rectangular)</strong></td>
<td></td>
</tr>
<tr>
<td>xGEBAK</td>
<td>Forms the right or left eigenvectors of a general matrix by backward transformation on the computed eigenvectors of the balanced matrix output by xGEBAL.</td>
</tr>
<tr>
<td>xGEBAL</td>
<td>Balances a general matrix.</td>
</tr>
<tr>
<td>xGEBRD</td>
<td>Reduces a general matrix to upper or lower bidiagonal form by an orthogonal transformation.</td>
</tr>
<tr>
<td>xGECON</td>
<td>Estimates the reciprocal of the condition number of a general matrix, using the factorization computed by xGEBTRF.</td>
</tr>
</tbody>
</table>
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGEEQU</td>
<td>Computes row and column scalings intended to equilibrate a general</td>
</tr>
<tr>
<td></td>
<td>rectangular matrix and reduce its condition number.</td>
</tr>
<tr>
<td>xGEES</td>
<td>Computes the eigenvalues and Schur factorization of a general matrix</td>
</tr>
<tr>
<td></td>
<td>(simple driver).</td>
</tr>
<tr>
<td>xGEESX</td>
<td>Computes the eigenvalues and Schur factorization of a general matrix</td>
</tr>
<tr>
<td></td>
<td>(expert driver).</td>
</tr>
<tr>
<td>xGEEV</td>
<td>Computes the eigenvalues and left and right eigenvectors of a general</td>
</tr>
<tr>
<td></td>
<td>matrix.</td>
</tr>
<tr>
<td>xGEEVX</td>
<td>Computes the eigenvalues and left and right eigenvectors of a general</td>
</tr>
<tr>
<td></td>
<td>matrix (simple driver).</td>
</tr>
<tr>
<td>xGEGS</td>
<td>Depreciated routine replaced by xGGES.</td>
</tr>
<tr>
<td>xGEGV</td>
<td>Depreciated routine replaced by xGGEV.</td>
</tr>
<tr>
<td>xGEHRD</td>
<td>Reduces a general matrix to upper Hessenberg form by an orthogonal</td>
</tr>
<tr>
<td></td>
<td>similarity transformation.</td>
</tr>
<tr>
<td>xGELQF</td>
<td>Computes LQ factorization of a general rectangular matrix.</td>
</tr>
<tr>
<td></td>
<td>(P)</td>
</tr>
<tr>
<td>xGELS</td>
<td>Computes the least squares solution to an over-determined system of</td>
</tr>
<tr>
<td></td>
<td>linear equations using a QR or LQ factorization of A.</td>
</tr>
<tr>
<td>xGELSD</td>
<td>Computes the least squares solution to an over-determined system of</td>
</tr>
<tr>
<td></td>
<td>linear equations using a divide and conquer method using a QR or LQ</td>
</tr>
<tr>
<td></td>
<td>factorization of A.</td>
</tr>
<tr>
<td>xGELSS</td>
<td>Computes the minimum-norm solution to a linear least squares problem</td>
</tr>
<tr>
<td></td>
<td>by using the SVD of a general rectangular matrix (simple driver).</td>
</tr>
<tr>
<td>xGELSY</td>
<td>Deprecated routine replaced by xGELS.</td>
</tr>
<tr>
<td>xGELSD</td>
<td>Computes the minimum-norm solution to a linear least squares problem</td>
</tr>
<tr>
<td></td>
<td>using a complete orthogonal factorization.</td>
</tr>
<tr>
<td>xGEQLF</td>
<td>Computes QL factorization of a general rectangular matrix.</td>
</tr>
<tr>
<td></td>
<td>(P)</td>
</tr>
<tr>
<td>xGEQP3</td>
<td>Computes QR factorization of general rectangular matrix using Level 3</td>
</tr>
<tr>
<td></td>
<td>BLAS.</td>
</tr>
<tr>
<td>xGEQPF</td>
<td>Deprecated routine replaced by xGEQP3.</td>
</tr>
<tr>
<td>xGEQRF</td>
<td>Computes QR factorization of a general rectangular matrix.</td>
</tr>
<tr>
<td></td>
<td>(P)</td>
</tr>
<tr>
<td>xGERFS</td>
<td>Refines solution to a system of linear equations.</td>
</tr>
<tr>
<td>xGERQF</td>
<td>Computes RQ factorization of a general rectangular matrix.</td>
</tr>
<tr>
<td></td>
<td>(P)</td>
</tr>
<tr>
<td>xGESDD</td>
<td>Computes SVD of general rectangular matrix using a divide and conquer</td>
</tr>
<tr>
<td></td>
<td>method.</td>
</tr>
<tr>
<td>xGESV</td>
<td>Solves a general system of linear equations (simple driver).</td>
</tr>
<tr>
<td>Routine</td>
<td>Function</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xGESVX</td>
<td>Solves a general system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xGESVD</td>
<td>Computes SVD of general rectangular matrix.</td>
</tr>
<tr>
<td>xGETRF (P)</td>
<td>Computes an LU factorization of a general rectangular matrix using partial pivoting with row interchanges.</td>
</tr>
<tr>
<td>xGETRI</td>
<td>Computes inverse of a general matrix using the factorization computed by xGETRF.</td>
</tr>
<tr>
<td>xGETRS (P)</td>
<td>Solves a general system of linear equations using the factorization computed by xGETRF.</td>
</tr>
</tbody>
</table>

**General Matrix-Generalized Problem (Pair of General Matrices)**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGGBAK</td>
<td>Forms the right or left eigenvectors of a generalized eigenvalue problem based on the output by xGGBAL.</td>
</tr>
<tr>
<td>xGGBAL</td>
<td>Balances a pair of general matrices for the generalized eigenvalue problem.</td>
</tr>
<tr>
<td>xGGES</td>
<td>Computes the generalized eigenvalues, Schur form, and left and/or right Schur vectors for two nonsymmetric matrices.</td>
</tr>
<tr>
<td>xGGESX</td>
<td>Computes the generalized eigenvalues, Schur form, and left and/or right Schur vectors.</td>
</tr>
<tr>
<td>xGGEV</td>
<td>Computes the generalized eigenvalues and the left and/or right generalized eigenvalues for two nonsymmetric matrices.</td>
</tr>
<tr>
<td>xGGEVX</td>
<td>Computes the generalized eigenvalues and the left and/or right generalized eigenvectors.</td>
</tr>
<tr>
<td>xGGGLM</td>
<td>Solves the GLM (Generalized Linear Regression Model) using the GQR (Generalized QR) factorization.</td>
</tr>
<tr>
<td>xGGHRD</td>
<td>Reduces two matrices to generalized upper Hessenberg form using orthogonal transformations.</td>
</tr>
<tr>
<td>xGGLSE</td>
<td>Solves the LSE (Constrained Linear Least Squares Problem) using the GRQ (Generalized RQ) factorization.</td>
</tr>
<tr>
<td>xGGQRF</td>
<td>Computes generalized QR factorization of two matrices.</td>
</tr>
<tr>
<td>xGGRQF</td>
<td>Computes generalized RQ factorization of two matrices.</td>
</tr>
<tr>
<td>xGGSVD</td>
<td>Computes the generalized singular value decomposition.</td>
</tr>
<tr>
<td>xGGSVPG</td>
<td>Computes an orthogonal or unitary matrix as a preprocessing step for calculating the generalized singular value decomposition.</td>
</tr>
</tbody>
</table>

**General Tridiagonal Matrix**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGTCON</td>
<td>Estimates the reciprocal of the condition number of a tridiagonal matrix, using the LU factorization as computed by xGTTTRF.</td>
</tr>
<tr>
<td>xGTRFS</td>
<td>Refines solution to a general tridiagonal system of linear equations.</td>
</tr>
<tr>
<td>Routine</td>
<td>Function</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xGTSV</td>
<td>Solves a general tridiagonal system of linear equations (simple driver).</td>
</tr>
<tr>
<td>xGTSVX</td>
<td>Solves a general tridiagonal system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xGTTRF</td>
<td>Computes an LU factorization of a general tridiagonal matrix using partial pivoting and row exchanges.</td>
</tr>
<tr>
<td>xGTTRS (P)</td>
<td>Solves general tridiagonal system of linear equations using the factorization computed by x.</td>
</tr>
</tbody>
</table>

**Hermitian Band Matrix**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHBEV or ZHBEV</td>
<td>Computes all eigenvalues and eigenvectors of a Hermitian band matrix.</td>
</tr>
<tr>
<td>CHBEVX or ZHBEV</td>
<td>Computes selected eigenvalues and eigenvectors of a Hermitian band matrix.</td>
</tr>
<tr>
<td>CHBGST or ZHBGST</td>
<td>Reduces Hermitian-definite banded generalized eigenproblem to standard form.</td>
</tr>
<tr>
<td>CHBGV or ZHBGV</td>
<td>Computes all eigenvalues and eigenvectors of a generalized Hermitian-definite banded eigenproblem.</td>
</tr>
<tr>
<td>CHBGVX or ZHBGVX</td>
<td>Computes selected eigenvalues and eigenvectors of a generalized Hermitian-definite banded eigenproblem.</td>
</tr>
<tr>
<td>CHBTRD or ZHBTRD</td>
<td>Reduces Hermitian band matrix to real symmetric tridiagonal form by using a unitary similarity transform.</td>
</tr>
</tbody>
</table>

**Hermitian Matrix**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECON or ZHECON</td>
<td>Estimates the reciprocal of the condition number of a Hermitian matrix using the factorization computed by CHETRF or ZHETRF.</td>
</tr>
<tr>
<td>CHEEV or ZHEEV</td>
<td>Computes all eigenvalues and eigenvectors of a Hermitian matrix (simple driver).</td>
</tr>
<tr>
<td>CHEEVD or ZHEEVD</td>
<td>Computes all eigenvalues and eigenvectors of a Hermitian matrix and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>CHEEVR or ZHEEVR</td>
<td>Computes selected eigenvalues and the eigenvectors of a complex Hermitian matrix.</td>
</tr>
</tbody>
</table>
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHEEVE or ZHEEVE</td>
<td>Computes selected eigenvalues and eigenvectors of a Hermitian matrix (expert driver).</td>
</tr>
<tr>
<td>CHEGST or ZHEGST</td>
<td>Reduces a Hermitian-definite generalized eigenproblem to standard form using the factorization computed by CPOTRF or ZPOTRF.</td>
</tr>
<tr>
<td>CHEGG or ZHEGG</td>
<td>Computes all the eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem. (Replacement with newer version CHEGVD or ZHEGVD suggested)</td>
</tr>
<tr>
<td>CHEGVD or ZHEGVD</td>
<td>Computes all the eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>CHEGV or ZHEGV</td>
<td>Computes selected eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem.</td>
</tr>
<tr>
<td>CHEGVD or ZHEGVD</td>
<td>Computes all the eigenvalues and eigenvectors of a complex generalized Hermitian-definite eigenproblem and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>CHERPFS or ZHERPS</td>
<td>Improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite.</td>
</tr>
<tr>
<td>CHESV or ZHESV</td>
<td>Solves a complex Hermitian indefinite system of linear equations (simple driver).</td>
</tr>
<tr>
<td>CHESVX or ZHESVX</td>
<td>Solves a complex Hermitian indefinite system of linear equations (simple driver).</td>
</tr>
<tr>
<td>CHETRD or ZHETRD</td>
<td>Reduces a Hermitian matrix to real symmetric tridiagonal form by using a unitary similarity transformation.</td>
</tr>
<tr>
<td>CHETRF or ZHETRF</td>
<td>Computes the factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.</td>
</tr>
<tr>
<td>CHETRI or ZHETR</td>
<td>Computes the inverse of a complex Hermitian indefinite matrix, using the factorization computed by CHETRF or ZHETRF.</td>
</tr>
<tr>
<td>CHETRP or ZHETRP</td>
<td>Solves a complex Hermitian indefinite matrix, using the factorization computed by CHETRF or ZHETRF.</td>
</tr>
</tbody>
</table>

**Hermitian Matrix in Packed Storage**

| CHPEV or ZHPEV | Computes all the eigenvalues and eigenvectors of a Hermitian matrix in packed storage (simple driver). (Replacement with newer version CHPEVD or ZHPEVD suggested) |
| 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. |

---

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHPCON or ZHPCON</td>
<td>Estimates the reciprocal of the condition number of a Hermitian indefinite matrix in packed storage using the factorization computed by CHPTRF or ZHPTRF.</td>
</tr>
<tr>
<td>CHPEVX or ZHPEVX</td>
<td>Computes selected eigenvalues and eigenvectors of a Hermitian matrix in packed storage (expert driver).</td>
</tr>
<tr>
<td>CHPEVD or ZHPEVD</td>
<td>Computes all the eigenvalues and eigenvectors of a Hermitian matrix in packed storage and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
</tbody>
</table>
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.

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 (P) or ZHPTRS (P)

Solves a complex Hermitian indefinite matrix in packed storage, using the factorization computed by CHPTRF or ZHPTRF.

Upper Hessenberg Matrix

xHSEIN

Computes right and/or left eigenvectors of upper Hessenberg matrix using inverse iteration.

xHSEQR

Computes eigenvectors and Shur factorization of upper Hessenberg matrix using multishift QR algorithm.

Upper Hessenberg Matrix-Generalized Problem (Hessenberg and Triangular Matrix)

xHGEQZ

Implements single-/double-shift version of QZ method for finding the generalized eigenvalues of the equation det(A - w(i) * B) = 0.
### Real Orthogonal Matrix in Packed Storage

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOPGTR or DOPGTR</td>
<td>Generates an orthogonal transformation matrix from a tridiagonal matrix determined by SSPTRD or DSPTRD.</td>
</tr>
<tr>
<td>SOPMTR or DOPMTR</td>
<td>Multiplies a general matrix by the orthogonal transformation matrix reduced to tridiagonal form by SSPTRD or DSPTRD.</td>
</tr>
</tbody>
</table>

### Real Orthogonal Matrix

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORBGR or DORGBR</td>
<td>Generates the orthogonal transformation matrices from reduction to bidiagonal form, as determined by SGEBRD or DGEBRD.</td>
</tr>
<tr>
<td>SORGRHR or DORGRHR</td>
<td>Generates the orthogonal transformation matrix reduced to Hessenberg form, as determined by SGEHRD or DGEHRD.</td>
</tr>
<tr>
<td>SORGLQ or DORGLQ</td>
<td>Generates an orthogonal matrix Q from an LQ factorization, as returned by SGEQLF or DGEQLF.</td>
</tr>
<tr>
<td>SORGQL or DORGQL</td>
<td>Generates an orthogonal matrix Q from a QR factorization, as returned by SGEQRF or DGEQRF.</td>
</tr>
<tr>
<td>SORGTR or DORGTR</td>
<td>Generates an orthogonal matrix reduced to tridiagonal form by SSYTRD or DSYTRD.</td>
</tr>
<tr>
<td>SORMBR or DORMBR</td>
<td>Multiplies a general matrix with the orthogonal matrix reduced to bidiagonal form, as determined by SGEBRD or DGEBRD.</td>
</tr>
<tr>
<td>SORMHR or DORMHR</td>
<td>Multiplies a general matrix by the orthogonal matrix reduced to Hessenberg form by SGEHRD or DGEHRD.</td>
</tr>
<tr>
<td>SORMLQ (P) or DORMLQ (P)</td>
<td>Multiplies a general matrix by the orthogonal matrix from an LQ factorization, as returned by SGEQLF or DGEQLF.</td>
</tr>
<tr>
<td>SORMQL (P) or DORMQL (P)</td>
<td>Multiplies a general matrix by the orthogonal matrix from a QL factorization, as returned by SGEQLF or DGEQLF.</td>
</tr>
<tr>
<td>SORMQR (P) or DORMQR (P)</td>
<td>Multiplies a general matrix by the orthogonal matrix from a QR factorization, as returned by SGEQRF or DGEQRF.</td>
</tr>
<tr>
<td>SORMR3 or DORMR3</td>
<td>Multiplies a general matrix by the orthogonal matrix returned by STZRZF or DTZRZF.</td>
</tr>
<tr>
<td>SORMRQ (P) or DORMRQ (P)</td>
<td>Multiplies a general matrix by the orthogonal matrix from an RQ factorization returned by SGERQF or DGERQF.</td>
</tr>
<tr>
<td>SORMRZ or DORMRZ</td>
<td>Multiplies a general matrix by the orthogonal matrix from an RZ factorization, as returned by STZRZF or DTZRZF.</td>
</tr>
</tbody>
</table>
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORMTR or DORMTR</td>
<td>Multiplies a general matrix by the orthogonal transformation matrix reduced to tridiagonal form by SSYTRD or DSYTRD.</td>
</tr>
<tr>
<td><strong>Symmetric or Hermitian Positive Definite Band Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>xPBCON</td>
<td>Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite band matrix, using the Cholesky factorization returned by xPBTRF.</td>
</tr>
<tr>
<td>xPBEQU</td>
<td>Computes equilibration scale factors for a symmetric or Hermitian positive definite band matrix.</td>
</tr>
<tr>
<td>xPBRFS</td>
<td>Refines solution to a symmetric or Hermitian positive definite banded system of linear equations.</td>
</tr>
<tr>
<td>xPBSTF</td>
<td>Computes a split Cholesky factorization of a real symmetric positive definite band matrix.</td>
</tr>
<tr>
<td>xPBSV</td>
<td>Solves a symmetric or Hermitian positive definite banded system of linear equations (simple driver).</td>
</tr>
<tr>
<td>xPBSVX</td>
<td>Solves a symmetric or Hermitian positive definite banded system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xPBTRF</td>
<td>Computes Cholesky factorization of a symmetric or Hermitian positive definite band matrix.</td>
</tr>
<tr>
<td>xPBTRS (P)</td>
<td>Solves symmetric positive definite banded matrix, using the Cholesky factorization computed by xPBTRF.</td>
</tr>
<tr>
<td><strong>Symmetric or Hermitian Positive Definite Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>xPOCON</td>
<td>Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite matrix, using the Cholesky factorization returned by xPOTRF.</td>
</tr>
<tr>
<td>xPOEQU</td>
<td>Computes equilibration scale factors for a symmetric or Hermitian positive definite matrix.</td>
</tr>
<tr>
<td>xPORFS</td>
<td>Refines solution to a linear system in a Cholesky-factored symmetric or Hermitian positive definite matrix.</td>
</tr>
<tr>
<td>xPOSV</td>
<td>Solves a symmetric or Hermitian positive definite system of linear equations (simple driver).</td>
</tr>
<tr>
<td>xPOSVX</td>
<td>Solves a symmetric or Hermitian positive definite system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xPOTRF (P)</td>
<td>Computes Cholesky factorization of a symmetric or Hermitian positive definite matrix.</td>
</tr>
<tr>
<td>xPOTRI</td>
<td>Computes the inverse of a symmetric or Hermitian positive definite matrix using the Cholesky-factorization returned by xPOTRF.</td>
</tr>
</tbody>
</table>
### Table A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xPOTRS</td>
<td>(P) Solves a symmetric or Hermitian positive definite system of linear equations, using the Cholesky factorization returned by xPOTRF.</td>
</tr>
<tr>
<td><strong>Symmetric or Hermitian Positive Definite Matrix in Packed Storage</strong></td>
<td></td>
</tr>
<tr>
<td>xPPCON</td>
<td>Reciprocal condition number of a Cholesky-factored symmetric positive definite matrix in packed storage.</td>
</tr>
<tr>
<td>xPPEQU</td>
<td>Computes equilibration scale factors for a symmetric or Hermitian positive definite matrix in packed storage.</td>
</tr>
<tr>
<td>xPPRFS</td>
<td>Refines solution to a linear system in a Cholesky-factored symmetric or Hermitian positive definite matrix in packed storage.</td>
</tr>
<tr>
<td>xPPSV</td>
<td>Solves a linear system in a symmetric or Hermitian positive definite matrix in packed storage (simple driver).</td>
</tr>
<tr>
<td>xPPSVX</td>
<td>Solves a linear system in a symmetric or Hermitian positive definite matrix in packed storage (expert driver).</td>
</tr>
<tr>
<td>xPPTRF</td>
<td>Computes Cholesky factorization of a symmetric or Hermitian positive definite matrix in packed storage.</td>
</tr>
<tr>
<td>xPPTRI</td>
<td>Computes the inverse of a symmetric or Hermitian positive definite matrix in packed storage using the Cholesky-factorization returned by xPPTRF.</td>
</tr>
<tr>
<td>xPPTRS</td>
<td>(P) 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 xPPTRF.</td>
</tr>
<tr>
<td><strong>Symmetric or Hermitian Positive Definite Tridiagonal Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>xPTCON</td>
<td>Estimates the reciprocal of the condition number of a symmetric or Hermitian positive definite tridiagonal matrix using the Cholesky factorization returned by xPTTRF.</td>
</tr>
<tr>
<td>xPTEQR</td>
<td>Computes all eigenvectors and eigenvalues of a real symmetric or Hermitian positive definite system of linear equations.</td>
</tr>
<tr>
<td>xPTRFS</td>
<td>Refines solution to a symmetric or Hermitian positive definite tridiagonal system of linear equations.</td>
</tr>
<tr>
<td>xPTSV</td>
<td>Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations (simple driver).</td>
</tr>
<tr>
<td>xPTSVX</td>
<td>Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations (expert driver).</td>
</tr>
<tr>
<td>xPTTRF</td>
<td>Computes the LDL(^{T}) factorization of a symmetric or Hermitian positive definite tridiagonal matrix.</td>
</tr>
<tr>
<td>xPTTRS</td>
<td>(P) Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations using the LDL(^{T}) factorization returned by xPTTRF.</td>
</tr>
<tr>
<td>Routine</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Real Symmetric Band Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>SSBEV or DSBEV</td>
<td>Computes all eigenvalues and eigenvectors of a symmetric band matrix.</td>
</tr>
<tr>
<td>SSBEVD or DSBEVD</td>
<td>Computes all eigenvalues and eigenvectors of a symmetric band matrix and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>SSBEVX or DSBEVX</td>
<td>Computes selected eigenvalues and eigenvectors of a symmetric band matrix.</td>
</tr>
<tr>
<td>SSBGST or DSBGST</td>
<td>Reduces symmetric-definite banded generalized eigenproblem to standard form.</td>
</tr>
<tr>
<td>SSBGV or DSBGV</td>
<td>Computes all eigenvalues and eigenvectors of a generalized symmetric-definite banded eigenproblem.</td>
</tr>
<tr>
<td>SSBGVX or DSBGVX</td>
<td>Computes selected eigenvalues and eigenvectors of a generalized symmetric-definite banded eigenproblem.</td>
</tr>
<tr>
<td>SSBGVD or DSBGVD</td>
<td>Computes all eigenvalues and eigenvectors of generalized symmetric-definite banded eigenproblem and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>SSBGVDX or DSBGVDX</td>
<td>Computes selected eigenvalues and eigenvectors of a generalized symmetric-definite banded eigenproblem.</td>
</tr>
<tr>
<td>SSBTRD or DSBTRD</td>
<td>Reduces symmetric band matrix to real symmetric tridiagonal form by using an orthogonal similarity transform.</td>
</tr>
<tr>
<td><strong>Symmetric Matrix in Packed Storage</strong></td>
<td></td>
</tr>
<tr>
<td>XSPCON</td>
<td>Estimates the reciprocal of the condition number of a symmetric packed matrix using the factorization computed by XSPTRF.</td>
</tr>
<tr>
<td>SSPEV or DSPEV</td>
<td>Computes all the eigenvalues and eigenvectors of a symmetric matrix in packed storage (simple driver).</td>
</tr>
<tr>
<td>SSPEVX or DSPEVX</td>
<td>Computes selected eigenvalues and eigenvectors of a symmetric matrix in packed storage (expert driver).</td>
</tr>
<tr>
<td>SSPEVD or DSPEVD</td>
<td>Computes all the eigenvalues and eigenvectors of a symmetric matrix in packed storage and uses a divide and conquer method to calculate eigenvectors.</td>
</tr>
<tr>
<td>SSPGST or DSPGST</td>
<td>Reduces a real symmetric-definite generalized eigenproblem to standard form where the coefficient matrices are in packed storage and uses the factorization computed by SPTRF or DPTRF.</td>
</tr>
<tr>
<td>SSPGVD or DSPGVD</td>
<td>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.</td>
</tr>
</tbody>
</table>
TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPGV or DSPGV</td>
<td>(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).</td>
</tr>
<tr>
<td>SSPGVX or DSPGVX</td>
<td>Computes selected eigenvalues and eigenvectors of a real generalized symmetric-definite eigenproblem where the coefficient matrices are in packed storage (expert driver).</td>
</tr>
<tr>
<td>xSPRFS</td>
<td>Improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite in packed storage.</td>
</tr>
<tr>
<td>xSPSV</td>
<td>Computes the solution to a system of linear equations where the coefficient matrix is a symmetric matrix in packed storage (simple driver).</td>
</tr>
<tr>
<td>xSPSVX</td>
<td>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).</td>
</tr>
<tr>
<td>SSPTRD or DSPTRD</td>
<td>Reduces a real symmetric matrix stored in packed form to real symmetric tridiagonal form using an orthogonal similarity transform.</td>
</tr>
<tr>
<td>xSPTRF</td>
<td>Computes the factorization of a symmetric packed matrix using the Bunch-Kaufman diagonal pivoting method.</td>
</tr>
<tr>
<td>xSPTRI</td>
<td>Computes the inverse of a symmetric indefinite matrix in packed storage using the factorization computed by xSPTRF.</td>
</tr>
<tr>
<td>xSPTRS (P)</td>
<td>Solves a system of linear equations by the symmetric matrix stored in packed format using the factorization computed by xSPTRF.</td>
</tr>
</tbody>
</table>

Real Symmetric Tridiagonal Matrix

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSTEBZ or DSTEBZ</td>
<td>Computes the eigenvalues of a real symmetric tridiagonal matrix.</td>
</tr>
<tr>
<td>xSTEDC</td>
<td>Computes all the eigenvalues and eigenvectors of a symmetric tridiagonal matrix using a divide and conquer method.</td>
</tr>
<tr>
<td>xSTEGR</td>
<td>Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using Relatively Robust Representations.</td>
</tr>
<tr>
<td>xSTEIN</td>
<td>Computes selected eigenvectors of a real symmetric tridiagonal matrix using inverse iteration.</td>
</tr>
<tr>
<td>xSTEQR</td>
<td>Computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using the implicit QL or QR algorithm.</td>
</tr>
<tr>
<td>SSTERF or DSTERF</td>
<td>Computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix using a root-free QL or QR algorithm variant.</td>
</tr>
<tr>
<td>SSTEV or DSTEV</td>
<td>(Replacement with newer version SSTEV or DSTEV suggested) Computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix (simple driver).</td>
</tr>
</tbody>
</table>
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.

xSTSV
Computes the solution to a system of linear equations where the coefficient matrix is a symmetric tridiagonal matrix.

xSTTRF
Computes the factorization of a symmetric tridiagonal matrix.

xSTTRS (P)
Computes the solution to a system of linear equations where the coefficient matrix is a symmetric tridiagonal matrix.

Symmetric Matrix

xSYCON
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.

SSYGX or DSYGX
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.

xSYRFS
Improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite.

xSYSV
Solves a real symmetric indefinite system of linear equations (simple driver).
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xSYSVX</td>
<td>Solves a real symmetric indefinite system of linear equations (expert driver).</td>
</tr>
<tr>
<td>SSYTRD or DSYTRD</td>
<td>Reduces a symmetric matrix to real symmetric tridiagonal form by using a orthogonal similarity transformation.</td>
</tr>
<tr>
<td>xSYTRF</td>
<td>Computes the factorization of a real symmetric indefinite matrix using the diagonal pivoting method.</td>
</tr>
<tr>
<td>xSYTRI</td>
<td>Computes the inverse of a symmetric indefinite matrix using the factorization computed by xSYTRF.</td>
</tr>
<tr>
<td>xSYTRS (P)</td>
<td>Solves a system of linear equations by the symmetric matrix using the factorization computed by xSYTRF.</td>
</tr>
</tbody>
</table>

**Triangular Band Matrix**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTBCON</td>
<td>Estimates the reciprocal condition number of a triangular band matrix.</td>
</tr>
<tr>
<td>xTBRFS</td>
<td>Determines error bounds and estimates for solving a triangular banded system of linear equations.</td>
</tr>
<tr>
<td>xTBTRS (P)</td>
<td>Solves a triangular banded system of linear equations.</td>
</tr>
</tbody>
</table>

**Triangular Matrix-Generalized Problem (Pair of Triangular Matrices)**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTGEVC</td>
<td>Computes right and/or left generalized eigenvectors of two upper triangular matrices.</td>
</tr>
<tr>
<td>xTGEXC</td>
<td>Reorders the generalized Schur decomposition of a real or complex matrix pair using an orthogonal or unitary equivalence transformation.</td>
</tr>
<tr>
<td>xTGSEN</td>
<td>Reorders the generalized real-Schur or Schur decomposition of two matrixes and computes the generalized eigenvalues.</td>
</tr>
<tr>
<td>xTGSJA</td>
<td>Computes the generalized SVD from two upper triangular matrices obtained from xGGSVP.</td>
</tr>
<tr>
<td>xTGSNA</td>
<td>Estimates reciprocal condition numbers for specified eigenvalues and eigenvectors of two matrices in real-Schur or Schur canonical form.</td>
</tr>
<tr>
<td>xTGSYL</td>
<td>Solves the generalized Sylvester equation.</td>
</tr>
</tbody>
</table>

**Triangular Matrix in Packed Storage**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTPCON</td>
<td>Estimates the reciprocal or the condition number of a triangular matrix in packed storage.</td>
</tr>
<tr>
<td>xTPRFS</td>
<td>Determines error bounds and estimates for solving a triangular system of linear equations where the coefficient matrix is in packed storage.</td>
</tr>
<tr>
<td>xTPTRI</td>
<td>Computes the inverse of a triangular matrix in packed storage.</td>
</tr>
<tr>
<td>xTPTRS (P)</td>
<td>Solves a triangular system of linear equations where the coefficient matrix is in packed storage.</td>
</tr>
</tbody>
</table>
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Triangular Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>XTRCON</td>
<td>Estimates the reciprocal or the condition number of a triangular matrix.</td>
</tr>
<tr>
<td>XTREVC</td>
<td>Computes right and/or left eigenvectors of an upper triangular matrix.</td>
</tr>
<tr>
<td>XTREXC</td>
<td>Reorders Schur factorization of matrix using an orthogonal or unitary similarity transformation.</td>
</tr>
<tr>
<td>XTRRFS</td>
<td>Determines error bounds and estimates for triangular system of a linear equations.</td>
</tr>
<tr>
<td>XTRSEN</td>
<td>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.</td>
</tr>
<tr>
<td>XTRSNA</td>
<td>Estimates the reciprocal condition numbers of selected eigenvalues and eigenvectors of an upper quasi-triangular matrix.</td>
</tr>
<tr>
<td>XTRSYL</td>
<td>Solves Sylvester matrix equation.</td>
</tr>
<tr>
<td>XTRTRI</td>
<td>Computes the inverse of a triangular matrix.</td>
</tr>
<tr>
<td>XTRTRS</td>
<td>(P) Solves a triangular system of linear equations.</td>
</tr>
<tr>
<td><strong>Trapezoidal Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>XTZRQF</td>
<td>Deprecated routine replaced by routine XZTRZF.</td>
</tr>
<tr>
<td>XZTRZF</td>
<td>Reduces a rectangular upper trapezoidal matrix to upper triangular form by means of orthogonal transformations.</td>
</tr>
<tr>
<td><strong>Unitary Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>CUNGBR or ZUNGBR</td>
<td>Generates the unitary transformation matrices from reduction to bidiagonal form, as determined by CGEBRD or ZGEBRD.</td>
</tr>
<tr>
<td>CUNGHR or ZUNGHR</td>
<td>Generates the orthogonal transformation matrix reduced to Hessenberg form, as determined by CGEHRD or ZGEBRD.</td>
</tr>
<tr>
<td>CUNGLQ or ZUNGLQ</td>
<td>Generates a unitary matrix Q from an LQ factorization, as returned by CGELQF or ZGELQF.</td>
</tr>
<tr>
<td>CUNGLQ or ZUNGLQ</td>
<td>Generates a unitary matrix Q from a QL factorization, as returned by CGEQLF or ZGEQLF.</td>
</tr>
<tr>
<td>CUNQR or ZUNQR</td>
<td>Generates a unitary matrix Q from a QR factorization, as returned by CGEQR2 or ZGEQR2.</td>
</tr>
<tr>
<td>CUNGQR or ZUNGQR</td>
<td>Generates a unitary matrix Q from an RQ factorization, as returned by CGEQRP or ZGEPQRP.</td>
</tr>
<tr>
<td>CUNGRQ or ZUNGRQ</td>
<td>Generates a unitary matrix reduced to tridiagonal form, by CHETRD or ZHETRD.</td>
</tr>
</tbody>
</table>
### TABLE A-1  LAPACK (Linear Algebra Package) Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUNMBR or ZUNMBR</td>
<td>Multiplies a general matrix with the unitary transformation matrix reduced to bidiagonal form, as determined by CGE BRD or ZGE BRD.</td>
</tr>
<tr>
<td>CUNMHR or ZUNMHR</td>
<td>Multiplies a general matrix by the unitary matrix reduced to Hessenberg form by CGEHRD or ZGEHRD.</td>
</tr>
<tr>
<td>CUNMLQ (P) or ZUNMLQ (P)</td>
<td>Multiplies a general matrix by the unitary matrix from an LQ factorization, as returned by CGELQF or ZGELQF.</td>
</tr>
<tr>
<td>CUNMQL (P) or ZUNMQL (P)</td>
<td>Multiplies a general matrix by the unitary matrix from a QL factorization, as returned by CGEQLF or ZGEQLF.</td>
</tr>
<tr>
<td>CUNMQR (P) or ZUNMQR (P)</td>
<td>Multiplies a general matrix by the unitary matrix from a QR factorization, as returned by CGEQRF or ZG EQRF.</td>
</tr>
<tr>
<td>CUNMRQ (P) or ZUNMRQ (P)</td>
<td>Multiplies a general matrix by the unitary matrix from an RQ factorization, as returned by CGERQF or ZGERQF.</td>
</tr>
<tr>
<td>CUNMRZ or ZUNMRZ</td>
<td>Multiplies a general matrix by the unitary matrix from an RZ factorization, as returned by CTZRZF or ZTZRZF.</td>
</tr>
<tr>
<td>CUNMTR or ZUNMTR</td>
<td>Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHETRD or ZHETRD.</td>
</tr>
</tbody>
</table>

#### Unitary Matrix in Packed Storage

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUPGTR or ZUPGTR</td>
<td>Generates the unitary transformation matrix from a tridiagonal matrix determined by CHPTRD or ZHPTRD.</td>
</tr>
<tr>
<td>CUPMTR or ZUPMTR</td>
<td>Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHPTRD or ZHPTRD.</td>
</tr>
</tbody>
</table>
## BLAS1 Routines

TABLE A-2 lists the Sun Performance Library BLAS1 routines. No Sun Performance Library BLAS1 routines are currently parallelized.

**TABLE A-2** BLAS1 (Basic Linear Algebra Subprograms, Level 1) Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASUM, DASUM, SCASUM, DZASUM</td>
<td>Sum of the absolute values of a vector</td>
</tr>
<tr>
<td>xAXPY</td>
<td>Product of a scalar and vector plus a vector</td>
</tr>
<tr>
<td>xCOPY</td>
<td>Copy a vector</td>
</tr>
<tr>
<td>SDOT, DDOT, DSDOT, SDDOT, CDOTU, ZDOTU, DQDOTA, DQDOTI</td>
<td>Dot product (inner product) Quad-precision DQDOTA, DQDOTI available only on SPARC</td>
</tr>
<tr>
<td>CDOTC, ZDOTC</td>
<td>Dot product conjugating first vector</td>
</tr>
<tr>
<td>SNRM2, DNRM2, SCNRM2, DZNRM2</td>
<td>Euclidean norm of a vector</td>
</tr>
<tr>
<td>xROTG</td>
<td>Set up Givens plane rotation</td>
</tr>
<tr>
<td>xROT, CSROT, ZDROT</td>
<td>Apply Givens plane rotation</td>
</tr>
<tr>
<td>SROTMG, DROTMG</td>
<td>Set up modified Givens’s plane rotation</td>
</tr>
<tr>
<td>SROTM, DROTM</td>
<td>Apply modified Givens’s rotation</td>
</tr>
<tr>
<td>ISAMAX, DAMAX, ICAMAX, IZAMAX</td>
<td>Index of element with maximum absolute value</td>
</tr>
<tr>
<td>xSCAL, CSSCAL, ZDSCAL</td>
<td>Scale a vector</td>
</tr>
<tr>
<td>xSWAP</td>
<td>Swap two vectors</td>
</tr>
<tr>
<td>CVMUL, ZVMUL</td>
<td>Compute scaled product of complex vectors</td>
</tr>
</tbody>
</table>
BLAS2 Routines

TABLE A-3 lists the Sun Performance Library BLAS2 routines. (P) denotes routines that are parallelized.

TABLE A-3  BLAS2 (Basic Linear Algebra Subprograms, Level 2) Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGBMV</td>
<td>Product of a matrix in banded storage and a vector</td>
</tr>
<tr>
<td>xGEMV (P)</td>
<td>Product of a general matrix and a vector</td>
</tr>
<tr>
<td>SGER (P), DGER (P), CGERC (P), ZGERC (P), CGERU (P), ZGERU (P)</td>
<td>Rank-1 update to a general matrix</td>
</tr>
<tr>
<td>CHBMV, ZHBMV</td>
<td>Product of a Hermitian matrix in banded storage and a vector</td>
</tr>
<tr>
<td>CHEMV (P), ZHEMV (P)</td>
<td>Product of a Hermitian matrix and a vector</td>
</tr>
<tr>
<td>CHER (P), ZHER (P)</td>
<td>Rank-1 update to a Hermitian matrix</td>
</tr>
<tr>
<td>CHER2, ZHER2</td>
<td>Rank-2 update to a Hermitian matrix</td>
</tr>
<tr>
<td>CHPMV (P), ZHPMV (P)</td>
<td>Product of a Hermitian matrix in packed storage and a vector</td>
</tr>
<tr>
<td>CHPR, ZHPR</td>
<td>Rank-1 update to a Hermitian matrix in packed storage</td>
</tr>
<tr>
<td>CHPR2, ZHPR2</td>
<td>Rank-2 update to a Hermitian matrix in packed storage</td>
</tr>
<tr>
<td>SSBMV, DSBMV</td>
<td>Product of a symmetric matrix in banded storage and a vector</td>
</tr>
<tr>
<td>SSPMV (P), DSPMV (P)</td>
<td>Product of a Symmetric matrix in packed storage and a vector</td>
</tr>
<tr>
<td>SSPR, DSPR</td>
<td>Rank-1 update to a real symmetric matrix in packed storage</td>
</tr>
<tr>
<td>SSPR2 (P), DSPR2 (P)</td>
<td>Rank-2 update to a real symmetric matrix in packed storage</td>
</tr>
<tr>
<td>SSYMV, (P) DSYMV (P)</td>
<td>Product of a symmetric matrix and a vector</td>
</tr>
<tr>
<td>SSYR (P), DSYR (P)</td>
<td>Rank-1 update to a real symmetric matrix</td>
</tr>
<tr>
<td>SSYR2 (P), DSYR2 (P)</td>
<td>Rank-2 update to a real symmetric matrix</td>
</tr>
<tr>
<td>xTBMV</td>
<td>Product of a triangular matrix in banded storage and a vector</td>
</tr>
<tr>
<td>xTBSV</td>
<td>Solution to a triangular system in banded storage of linear equations</td>
</tr>
<tr>
<td>xTPMV</td>
<td>Product of a triangular matrix in packed storage and a vector</td>
</tr>
<tr>
<td>xTPSV</td>
<td>Solution to a triangular system of linear equations in packed storage</td>
</tr>
<tr>
<td>xTRMV (P)</td>
<td>Product of a triangular matrix and a vector</td>
</tr>
<tr>
<td>xTRSV (P)</td>
<td>Solution to a triangular system of linear equations</td>
</tr>
</tbody>
</table>
BLAS3 Routines

TABLE A-4 lists the Sun Performance Library BLAS3 routines. (P) denotes routines that are parallelized.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGEMM (P)</td>
<td>Product of two general matrices</td>
</tr>
<tr>
<td>CHEMM (P) or ZHEMM (P)</td>
<td>Product of a Hermitian matrix and a general matrix</td>
</tr>
<tr>
<td>CHERK (P) or ZHERK (P)</td>
<td>Rank-k update of a Hermitian matrix</td>
</tr>
<tr>
<td>CHER2K (P) or ZHER2K (P)</td>
<td>Rank-2k update of a Hermitian matrix</td>
</tr>
<tr>
<td>SYMM (P)</td>
<td>Product of a symmetric matrix and a general matrix</td>
</tr>
<tr>
<td>SYRK (P)</td>
<td>Rank-k update of a symmetric matrix</td>
</tr>
<tr>
<td>SYR2K (P)</td>
<td>Rank-2k update of a symmetric matrix</td>
</tr>
<tr>
<td>TRMM (P)</td>
<td>Product of a triangular matrix and a general matrix</td>
</tr>
<tr>
<td>TRSM (P)</td>
<td>Solution for a triangular system of equations</td>
</tr>
</tbody>
</table>

Sparse BLAS Routines

TABLE A-5 lists the Sun Performance Library sparse BLAS routines. (P) denotes routines that are parallelized.

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAXPYI</td>
<td>Adds a scalar multiple of a sparse vector X to a full vector Y.</td>
</tr>
<tr>
<td>XBCOMM (P)</td>
<td>Block coordinate matrix-matrix multiply.</td>
</tr>
<tr>
<td>XBDI MM (P)</td>
<td>Block diagonal format matrix-matrix multiply.</td>
</tr>
<tr>
<td>XBDSM (P)</td>
<td>Block Diagonal format triangular solve.</td>
</tr>
<tr>
<td>XBEILMM (P)</td>
<td>Block Ellpack format matrix-matrix multiply.</td>
</tr>
<tr>
<td>XBEILSM (P)</td>
<td>Block Ellpack format triangular solve.</td>
</tr>
<tr>
<td>XBSCDM (P)</td>
<td>Block compressed sparse column format matrix-matrix multiply.</td>
</tr>
<tr>
<td>XBSCSM (P)</td>
<td>Block compressed sparse column format triangular solve.</td>
</tr>
<tr>
<td>Routines</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xBSRMM (P)</td>
<td>Block compressed sparse row format matrix-matrix multiply.</td>
</tr>
<tr>
<td>xBSRSM (P)</td>
<td>Block compressed sparse row format triangular solve.</td>
</tr>
<tr>
<td>xCOOMM (P)</td>
<td>Coordinate format matrix-matrix multiply.</td>
</tr>
<tr>
<td>xCSCMM (P)</td>
<td>Compressed sparse column format matrix-matrix multiply</td>
</tr>
<tr>
<td>xCSCSM (P)</td>
<td>Compressed sparse column format triangular solve</td>
</tr>
<tr>
<td>xCSRMM (P)</td>
<td>Compressed sparse row format matrix-matrix multiply</td>
</tr>
<tr>
<td>xCSRSM (P)</td>
<td>Compressed sparse row format triangular solve</td>
</tr>
<tr>
<td>xDIAMM (P)</td>
<td>Diagonal format matrix-matrix multiply.</td>
</tr>
<tr>
<td>xDIASM (P)</td>
<td>Diagonal format triangular solve.</td>
</tr>
<tr>
<td>SDOTI, DDTI,</td>
<td>Computes the dot product of a sparse vector and a full vector.</td>
</tr>
<tr>
<td>CDOTUI, or ZDOTUI</td>
<td></td>
</tr>
<tr>
<td>CDOTCI, or ZDOTCI</td>
<td>Computes the conjugate dot product of a sparse vector and a full vector.</td>
</tr>
<tr>
<td>xELLMM (P)</td>
<td>Ellpack format matrix-matrix multiply.</td>
</tr>
<tr>
<td>xELLSM (P)</td>
<td>Ellpack format triangular solve.</td>
</tr>
<tr>
<td>xCGTHR</td>
<td>Given a full vector, creates a sparse vector and corresponding index vector.</td>
</tr>
<tr>
<td>xCGTHRZ</td>
<td>Given a full vector, creates a sparse vector and corresponding index vector and zeros the full vector.</td>
</tr>
<tr>
<td>xJADMM (P)</td>
<td>Jagged diagonal matrix-matrix multiply.</td>
</tr>
<tr>
<td>SJADRP or DJADRP</td>
<td>Right permutation of a jagged diagonal matrix.</td>
</tr>
<tr>
<td>xJADSM (P)</td>
<td>Jagged diagonal triangular solve.</td>
</tr>
<tr>
<td>SROTI or DROTI</td>
<td>Applies a Givens rotation to a sparse vector and a full vector.</td>
</tr>
<tr>
<td>xCSCTR</td>
<td>Given a sparse vector and corresponding index vector, puts those elements into a full vector.</td>
</tr>
<tr>
<td>xSKYMM (P)</td>
<td>Skyline format matrix-matrix multiply.</td>
</tr>
<tr>
<td>xSKYSM (P)</td>
<td>Skyline format triangular solve.</td>
</tr>
<tr>
<td>xVBRMM (P)</td>
<td>Variable block sparse row format matrix-matrix multiply</td>
</tr>
<tr>
<td>xVBRSM (P)</td>
<td>Variable block sparse row format triangular solve.</td>
</tr>
</tbody>
</table>
Sparse Solver Routines

The following tables list routines from SPSOLVE and SuperLU sparse solvers in the Sun Performance Library. (P) denotes routines that are parallelized.

### TABLE A-6  SPSOLVE Routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGSSFS (P)</td>
<td>One call interface to SPSOLVE.</td>
</tr>
<tr>
<td>xGSSIN</td>
<td>SPSOLVE initialization.</td>
</tr>
<tr>
<td>xGSSOR</td>
<td>Fill reducing ordering and symbolic factorization.</td>
</tr>
<tr>
<td>xGSSFA (P)</td>
<td>Matrix value input and numeric factorization.</td>
</tr>
<tr>
<td>xGSSL</td>
<td>Triangular solve.</td>
</tr>
<tr>
<td>xGSSUO</td>
<td>Sets user-specified ordering permutation.</td>
</tr>
<tr>
<td>xGSSRP</td>
<td>Returns permutation used by solver.</td>
</tr>
<tr>
<td>xGSSCO</td>
<td>Returns condition number estimate of coefficient matrix.</td>
</tr>
<tr>
<td>xGSSDA</td>
<td>Deallocate SPSOLVE memory.</td>
</tr>
<tr>
<td>xGSSPS</td>
<td>Prints solver statistics.</td>
</tr>
</tbody>
</table>

### TABLE A-7  SuperLU Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xgstrf</td>
<td>Computes factorization</td>
</tr>
<tr>
<td>xgssvx</td>
<td>Factorizes and solves (expert driver)</td>
</tr>
<tr>
<td>xgssv</td>
<td>Factorizes and solves (simple driver)</td>
</tr>
<tr>
<td>xgstrs</td>
<td>Computes triangular solve</td>
</tr>
<tr>
<td>xgerfs</td>
<td>Improves computed solution; provides error bounds</td>
</tr>
<tr>
<td>xlangs</td>
<td>Computes one-norm, Frobenius-norm, or infinity-norm</td>
</tr>
<tr>
<td>xgequ</td>
<td>Computes row and column scalings</td>
</tr>
<tr>
<td>xgscon</td>
<td>Estimates reciprocal of condition number</td>
</tr>
<tr>
<td>xlaqgs</td>
<td>Equilibrates a general sparse matrix</td>
</tr>
<tr>
<td>LUSolveTime</td>
<td>Returns time spent in solve stage</td>
</tr>
<tr>
<td>LUFactTime</td>
<td>Returns time spent in factorization stage</td>
</tr>
<tr>
<td>LUFactFlops</td>
<td>Returns number of floating point operations in factorization stage</td>
</tr>
</tbody>
</table>
TABLE A-7  SuperLU Routines  *(Continued)*

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUSolveFlops</td>
<td>Returns number of floating point operations in solve stage</td>
</tr>
<tr>
<td>xQuerySpace</td>
<td>Returns information on the memory statistics</td>
</tr>
<tr>
<td>sp_ienv</td>
<td>Returns specified machine dependent parameter</td>
</tr>
<tr>
<td>xPrintPerf</td>
<td>Prints statistics collected by the computational routines</td>
</tr>
<tr>
<td>set_default_options</td>
<td>Sets parameters that control solver behavior to default options</td>
</tr>
<tr>
<td>StatInit</td>
<td>Allocates and initializes structure that stores performance statistics</td>
</tr>
<tr>
<td>StatFree</td>
<td>Frees structure that stores performance statistics</td>
</tr>
<tr>
<td>Destroy_Dense_Matrix</td>
<td>Deallocates a SuperMatrix in dense format</td>
</tr>
<tr>
<td>Destroy_SuperNode_Matrix</td>
<td>Deallocates a SuperMatrix in supernodal format</td>
</tr>
<tr>
<td>Destroy_CompCol_Matrix</td>
<td>Deallocates a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>Destroy_CompCol_Permuted</td>
<td>Deallocates a SuperMatrix in permuted compressed sparse column format</td>
</tr>
<tr>
<td>Destroy_SuperMatrix_Store</td>
<td>Deallocates actual storage that stores matrix in a SuperMatrix</td>
</tr>
<tr>
<td>xCopy_CompCol_Matrix</td>
<td>Copies a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>xCreate_CompCol_Matrix</td>
<td>Allocates a SuperMatrix in compressed sparse column format</td>
</tr>
<tr>
<td>xCreate_Dense_Matrix</td>
<td>Allocates a SuperMatrix in dense format</td>
</tr>
<tr>
<td>xCreate_CompRow_Matrix</td>
<td>Allocates a SuperMatrix in compressed sparse row format</td>
</tr>
<tr>
<td>xCreate_SuperNode_Matrix</td>
<td>Allocates a SuperMatrix in supernodal format</td>
</tr>
<tr>
<td>sp_preorder</td>
<td>Permutes columns of original sparse matrix</td>
</tr>
<tr>
<td>sp_sgemm</td>
<td>Multiplies a SuperMatrix by a dense matrix</td>
</tr>
<tr>
<td>sp_dgemm</td>
<td></td>
</tr>
<tr>
<td>sp_cgemm</td>
<td></td>
</tr>
<tr>
<td>sp_zgemm</td>
<td></td>
</tr>
</tbody>
</table>
Signal Processing Library Routines

Sun Performance Library contains routines for computing the fast Fourier transform, sine and cosine transforms, and convolution and correlation.

FFT Routines

Sun Performance Library provides a set of FFT interfaces that supersedes a subset of the FFTPACK and VFFTPACK routines provided in earlier Sun Performance Library releases. The old FFT interfaces are included for backward compatibility, and users are encouraged to use the new interfaces. For information on individual FFT routines, see the section 3P man pages.

TABLE A-8 shows the mapping between the Sun Performance Library FFT routines and the corresponding FFTPACK and VFFTPACK routines. (P) denotes routines that are parallelized.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Replaces</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFFTC (P)</td>
<td>CFFT1, CFFT2 (P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward or inverse FFT of a complex sequence.</td>
</tr>
<tr>
<td>CFFTC2 (P)</td>
<td>CFFT2I, CFFT2F (P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the two-dimensional forward or inverse FFT of a two-dimensional complex array.</td>
</tr>
<tr>
<td>CFFTC3 (P)</td>
<td>CFFT3I, CFFT3F (P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the three-dimensional forward or inverse FFT of three-dimensional complex array.</td>
</tr>
<tr>
<td>CFPTCM (P)</td>
<td>VCFFT1, VCFPTF (P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward or inverse FFT of a set of data sequences stored in a two-dimensional complex array.</td>
</tr>
<tr>
<td>CFPTS</td>
<td>RFFFT1, RFFFT2, EZFFT1, EZFFT2</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a complex sequence.</td>
</tr>
<tr>
<td>CFPTS2</td>
<td>RFFFT2I, RFFFT2B</td>
<td>Initialize the trigonometric weight and factor tables or compute the two-dimensional inverse FFT of a two-dimensional complex array.</td>
</tr>
<tr>
<td>CFPTS3 (P)</td>
<td>RFFFT3I, RFFFT3B</td>
<td>Initialize the trigonometric weight and factor tables or compute the three-dimensional inverse FFT of three-dimensional complex array.</td>
</tr>
<tr>
<td>Routine</td>
<td>Replaces</td>
<td>Function</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CFFTS</td>
<td>VRFFTIB(P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a set of data sequences stored in a two-dimensional complex array.</td>
</tr>
<tr>
<td>DFFTZ</td>
<td>DFPTI, DFFTIB</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a double precision sequence.</td>
</tr>
<tr>
<td>DFFTZ2</td>
<td>DFPT2I, DFPT2B</td>
<td>Initialize the trigonometric weight and factor tables or compute the two-dimensional forward FFT of a two-dimensional double precision array.</td>
</tr>
<tr>
<td>DFFTZ3(P)</td>
<td>DFPT3I, DFPT3B</td>
<td>Initialize the trigonometric weight and factor tables or compute the three-dimensional forward FFT of three-dimensional double precision array.</td>
</tr>
<tr>
<td>DFFT2M</td>
<td>VDFPTIB(P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a set of data sequences stored in a two-dimensional double precision array.</td>
</tr>
<tr>
<td>SFFTC</td>
<td>RFPTI, RFFTFB</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a real sequence.</td>
</tr>
<tr>
<td>SFFTC2</td>
<td>RFPT2I, RFPT2B</td>
<td>Initialize the trigonometric weight and factor tables or compute the two-dimensional forward FFT of a two-dimensional real array.</td>
</tr>
<tr>
<td>SFFTC3(P)</td>
<td>RFPT3I, RFPT3B</td>
<td>Initialize the trigonometric weight and factor tables or compute the three-dimensional forward FFT of three-dimensional real array.</td>
</tr>
<tr>
<td>SFFTSM</td>
<td>VRFFTIB(P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a set of data sequences stored in a two-dimensional real array.</td>
</tr>
<tr>
<td>ZFFTD</td>
<td>DFPTI, DFBTB</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a double complex sequence.</td>
</tr>
<tr>
<td>ZFFTD2</td>
<td>DFPT2I, DFPT2B</td>
<td>Initialize the trigonometric weight and factor tables or compute the two-dimensional inverse FFT of a two-dimensional double complex array.</td>
</tr>
<tr>
<td>ZFFTD3(P)</td>
<td>DFPT3I, DFPT3B</td>
<td>Initialize the trigonometric weight and factor tables or compute the three-dimensional inverse FFT of three-dimensional double complex array.</td>
</tr>
<tr>
<td>ZFFTSM</td>
<td>VDFPTIB(P)</td>
<td>Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a set of data sequences stored in a two-dimensional double complex array.</td>
</tr>
</tbody>
</table>
Fast Cosine and Sine Transforms

Sun Performance Library fast cosine and sine transform routines are based on the routines contained in FFTPACK (http://www.netlib.org/fftpack/). Routines with a V prefix are vectorized routines that are based on the routines contained in VFFTPACK (http://www.netlib.org/vfftpack/).

TABLE A-9 lists the Sun Performance Library sine and cosine transform routines.

TABLE A-9  Sine and Cosine Transform Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSQB, DCOSQB, VCOQB, VDCOSQB</td>
<td>Cosine quarter-wave synthesis.</td>
</tr>
<tr>
<td>COSQP, DCOSQE, VCOSQF, VDCOSQF</td>
<td>Cosine quarter-wave transform.</td>
</tr>
<tr>
<td>COSQI, DCOSQI, VCOSQI, VDCOSQI</td>
<td>Initialize cosine quarter-wave transform and synthesis.</td>
</tr>
<tr>
<td>COST, DCOST, VCOST, VDCOST</td>
<td>Cosine even-wave transform.</td>
</tr>
<tr>
<td>COSTI, DCOSTI, VCOSTI, VDCOSTI</td>
<td>Initialize cosine even-wave transform.</td>
</tr>
<tr>
<td>SINQB, DSINQB, VSINQB, VDSINQB</td>
<td>Sine quarter-wave synthesis.</td>
</tr>
</tbody>
</table>
Convolution and Correlation Routines

**TABLE A-10** lists the Sun Performance Library convolution and correlation routines.

**TABLE A-10** Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xCNVCOR</td>
<td>Computes convolution or correlation</td>
</tr>
<tr>
<td>xCNVCOR2</td>
<td>Computes two-dimensional convolution or correlation</td>
</tr>
</tbody>
</table>

Miscellaneous Signal Processing Routines

**TABLE A-11** lists the miscellaneous Sun Performance Library signal processing routines.

**TABLE A-11** Convolution and Correlation Routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFFTOPT, DFFTOPT, CFFTOPT, ZFFTOPT</td>
<td>Compute the length of the closest FFT</td>
</tr>
<tr>
<td>SWIENER or DWEINER</td>
<td>Performs Wiener deconvolution of two signals</td>
</tr>
<tr>
<td>XTRANS (P)</td>
<td>Transposes array</td>
</tr>
</tbody>
</table>
Interval BLAS (IBLAS) Routines

Sun Performance Library includes the interval BLAS routines listed in TABLE A-12, which operate on interval scalars, interval vectors, and interval matrices (dense, banded, symmetric, and triangular).

**TABLE A-12**  Interval BLAS Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>amax_val_i</td>
<td>Max absolute value and location.</td>
</tr>
<tr>
<td>amin_val_i</td>
<td>Min absolute value and location.</td>
</tr>
<tr>
<td>axphy_i</td>
<td>Scaled vector accumulation.</td>
</tr>
<tr>
<td>cancel_i</td>
<td>Scaled cancellation.</td>
</tr>
<tr>
<td>constructv_i</td>
<td>Constructs an interval vector.</td>
</tr>
<tr>
<td>copy_i</td>
<td>Interval vector copy.</td>
</tr>
<tr>
<td>disjv_i</td>
<td>Checks if two interval vectors disjoint.</td>
</tr>
<tr>
<td>dot_i</td>
<td>Scaled dot product of two interval vectors.</td>
</tr>
<tr>
<td>emptyelev_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>encv_i</td>
<td>Check if an interval vector is enclosed in another.</td>
</tr>
<tr>
<td>fpinfo_i</td>
<td>Environmental enquiry.</td>
</tr>
<tr>
<td>gbmv_i</td>
<td>Interval matrix-vector multiplication.</td>
</tr>
<tr>
<td>gb_acc_i</td>
<td>General band matrix accumulation and scale.</td>
</tr>
<tr>
<td>gb_add_i</td>
<td>General band matrix add and scale.</td>
</tr>
<tr>
<td>gb_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>gb_copy_i</td>
<td>General band interval matrix copy.</td>
</tr>
<tr>
<td>gb_diag_scale_i</td>
<td>Diagonal scaling of an interval matrix.</td>
</tr>
<tr>
<td>gb_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>gb_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>gb_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>gb_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>gb_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>gb_interiom_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>gb_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>gb_lrscale_i</td>
<td>Two-sided diagonal scaling.</td>
</tr>
<tr>
<td>gb_middm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
</tbody>
</table>
### TABLE A-12  Interval BLAS Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>gb_norm_i</td>
<td>General band interval matrix norms.</td>
</tr>
<tr>
<td>gb_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>gb_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>gb_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>gb_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>gemm_i</td>
<td>General interval matrix product.</td>
</tr>
<tr>
<td>gemv_i</td>
<td>General interval matrix and vector multiplication.</td>
</tr>
<tr>
<td>ger_i</td>
<td>Rank one update.</td>
</tr>
<tr>
<td>ge_acc_i</td>
<td>General matrix accumulation and scale.</td>
</tr>
<tr>
<td>ge_add_i</td>
<td>General interval matrix add and scale.</td>
</tr>
<tr>
<td>ge_constrctm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>ge_copy_i</td>
<td>General interval matrix copy.</td>
</tr>
<tr>
<td>ge_diag_scale_i</td>
<td>Diagonal scaling an interval matrix.</td>
</tr>
<tr>
<td>ge_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>ge_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>ge_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>ge_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>ge_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>ge_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>ge_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>ge_lrscale_i</td>
<td>Two-sided diagonal scaling.</td>
</tr>
<tr>
<td>ge_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>ge_norm_i</td>
<td>General interval matrix norms.</td>
</tr>
<tr>
<td>ge_permute_i</td>
<td>Permute an general interval matrix.</td>
</tr>
<tr>
<td>ge_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>ge_trans_i</td>
<td>Matrix transposition.</td>
</tr>
<tr>
<td>ge_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>ge_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>ge_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>hullv_i</td>
<td>Convex hull of an interval vector with another.</td>
</tr>
<tr>
<td>Routine</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>infv_i</td>
<td>The left endpoint of an interval vector.</td>
</tr>
<tr>
<td>interiorv_i</td>
<td>If an interval vector is in the interior of another.</td>
</tr>
<tr>
<td>interv_i</td>
<td>Intersection of an interval vector with another.</td>
</tr>
<tr>
<td>midv_i</td>
<td>The approximate midpoint of an interval vector.</td>
</tr>
<tr>
<td>norm_i</td>
<td>Interval vector norms.</td>
</tr>
<tr>
<td>permute_i</td>
<td>Permute interval vector.</td>
</tr>
<tr>
<td>rscale_i</td>
<td>Reciprocal scale of an interval vector.</td>
</tr>
<tr>
<td>sbmv_i</td>
<td>Interval symmetric matrix vector product.</td>
</tr>
<tr>
<td>sb_acc_i</td>
<td>Symmetric band matrix accumulation and scale.</td>
</tr>
<tr>
<td>sb_add_i</td>
<td>Symmetric band matrix add and scale.</td>
</tr>
<tr>
<td>sb_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>sb_copy_i</td>
<td>Symmetric band interval matrix copy.</td>
</tr>
<tr>
<td>sb_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>sb_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>sb_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>sb_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sb_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sb_interiom_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>sb_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>sb_lrscale_i</td>
<td>Two-sided diagonal scaling.</td>
</tr>
<tr>
<td>sb_mdm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>sb_norm_i</td>
<td>Symmetric band interval matrix norms.</td>
</tr>
<tr>
<td>sb_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sb_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sb_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>sb_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>spmv_i</td>
<td>Interval symmetric matrix vector product.</td>
</tr>
<tr>
<td>spr_i</td>
<td>Symmetric rank one update.</td>
</tr>
<tr>
<td>sp_acc_i</td>
<td>Symmetric packed matrix accumulation and scale.</td>
</tr>
<tr>
<td>sp_add_i</td>
<td>Symmetric packed matrix add and scale.</td>
</tr>
</tbody>
</table>
### TABLE A-12 Interval BLAS Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>sp_copy_i</td>
<td>Symmetric packed interval matrix copy.</td>
</tr>
<tr>
<td>sp_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>sp_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>sp_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>sp_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sp_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sp_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>sp_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>sp_lrscale_i</td>
<td>Two-sided diagonal scaling.</td>
</tr>
<tr>
<td>sp_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>sp_norm_i</td>
<td>Symmetric packed interval matrix norms.</td>
</tr>
<tr>
<td>sp_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sp_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sp_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>sp_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>sumsq_i</td>
<td>Sum of squares.</td>
</tr>
<tr>
<td>sum_i</td>
<td>Sum the entries of an interval vector.</td>
</tr>
<tr>
<td>supv_i</td>
<td>The right endpoint of an interval vector.</td>
</tr>
<tr>
<td>swap_i</td>
<td>Interval vector swap.</td>
</tr>
<tr>
<td>symm_i</td>
<td>Symmetric interval matrix product.</td>
</tr>
<tr>
<td>symv_i</td>
<td>Interval symmetric matrix vector product.</td>
</tr>
<tr>
<td>syr_i</td>
<td>Symmetric rank one update.</td>
</tr>
<tr>
<td>sy_acc_i</td>
<td>Symmetric interval matrix accumulation and scale.</td>
</tr>
<tr>
<td>sy_add_i</td>
<td>Symmetric matrix add and scale.</td>
</tr>
<tr>
<td>sy_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>sy_copy_i</td>
<td>Symmetric interval matrix copy.</td>
</tr>
<tr>
<td>sy_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>sy_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>sy_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
</tbody>
</table>
TABLE A-12  Interval BLAS Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sy_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sy_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sy_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>sy_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>sy_lrscale_i</td>
<td>Two-sided diagonal scaling.</td>
</tr>
<tr>
<td>sy_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>sy_norm_i</td>
<td>Symmetric interval matrix norms.</td>
</tr>
<tr>
<td>sy_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>sy_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>sy_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>sy_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>tbmv_i</td>
<td>Interval triangular matrix vector product.</td>
</tr>
<tr>
<td>tbsv_i</td>
<td>Interval triangular solve with a vector.</td>
</tr>
<tr>
<td>tb_acc_i</td>
<td>Matrix accumulation and scale.</td>
</tr>
<tr>
<td>tb_add_i</td>
<td>Triangular band matrix add and scale.</td>
</tr>
<tr>
<td>tb_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>tb_copy_i</td>
<td>Triangular band interval matrix copy.</td>
</tr>
<tr>
<td>tb_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>tb_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>tb_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>tb_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tb_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tb_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>tb_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>tb_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>tb_norm_i</td>
<td>Triangular band interval matrix norms.</td>
</tr>
<tr>
<td>tb_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tb_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tb_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>tb_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>Routine</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tpmv_i</td>
<td>Interval triangular matrix vector product.</td>
</tr>
<tr>
<td>tpsv_i</td>
<td>Interval triangular solve with a vector.</td>
</tr>
<tr>
<td>tp_acc_i</td>
<td>Matrix accumulation and scale.</td>
</tr>
<tr>
<td>tp_add_i</td>
<td>Triangular packed matrix add and scale.</td>
</tr>
<tr>
<td>tp_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>tp_copy_i</td>
<td>Triangular packed interval matrix copy.</td>
</tr>
<tr>
<td>tp_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>tp_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>tp_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
<tr>
<td>tp_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tp_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tp_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>tp_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>tp_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>tp_norm_i</td>
<td>Triangular packed interval matrix norms.</td>
</tr>
<tr>
<td>tp_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tp_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tp_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>tp_winterm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>trmm_i</td>
<td>Triangular interval matrix matrix product.</td>
</tr>
<tr>
<td>trmv_i</td>
<td>Interval triangular matrix vector product.</td>
</tr>
<tr>
<td>trsm_i</td>
<td>Interval triangular solve.</td>
</tr>
<tr>
<td>trav_i</td>
<td>Interval triangular solve with a vector.</td>
</tr>
<tr>
<td>tr_acc_i</td>
<td>Matrix accumulation and scale.</td>
</tr>
<tr>
<td>tr_add_i</td>
<td>Triangular matrix add and scale.</td>
</tr>
<tr>
<td>tr_constructm_i</td>
<td>Constructs an interval matrix from two floating point matrices.</td>
</tr>
<tr>
<td>tr_copy_i</td>
<td>Triangular interval matrix copy.</td>
</tr>
<tr>
<td>tr_disjm_i</td>
<td>If two interval matrices are disjoint.</td>
</tr>
<tr>
<td>tr_emptyelem_i</td>
<td>Empty entry and its location.</td>
</tr>
<tr>
<td>tr_encm_i</td>
<td>If an interval matrix is enclosed in another.</td>
</tr>
</tbody>
</table>
TABLE A-12  Interval BLAS Routines (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr_hullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tr_infm_i</td>
<td>Left endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tr_interiorm_i</td>
<td>If an interval matrix is in interior of another.</td>
</tr>
<tr>
<td>tr_interm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>tr_midm_i</td>
<td>Midpoint matrix of an interval matrix.</td>
</tr>
<tr>
<td>tr_norm_i</td>
<td>Triangular interval matrix norms.</td>
</tr>
<tr>
<td>tr_supm_i</td>
<td>Right endpoint of an interval matrix.</td>
</tr>
<tr>
<td>tr_whullm_i</td>
<td>Convex hull of two interval matrices.</td>
</tr>
<tr>
<td>tr_widthm_i</td>
<td>Elementwise width of an interval matrix.</td>
</tr>
<tr>
<td>tr_widthm_i</td>
<td>Intersection of two interval matrices.</td>
</tr>
<tr>
<td>waxpby_i</td>
<td>Scaled vector addition.</td>
</tr>
<tr>
<td>wcancel_i</td>
<td>Scaled cancellation.</td>
</tr>
<tr>
<td>whullv_i</td>
<td>Convex hull of an interval vector with another.</td>
</tr>
<tr>
<td>widthv_i</td>
<td>The elementwise width of an interval vector.</td>
</tr>
<tr>
<td>winterv_i</td>
<td>Intersection of an interval vector with another.</td>
</tr>
</tbody>
</table>

See the section 3P man pages for information on using each routine.

Sort Routines

TABLE A-13 lists the Sun Performance Library sort routines. (P) denotes routines that are parallelized on Solaris/SPARC platforms. All routines are single-threaded on Solaris/x86 platforms whether denoted by (P) or not.

TABLE A-13  Sort Routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS_DSORT  (P)</td>
<td>Sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm.</td>
</tr>
<tr>
<td>BLAS_DSORTV (P)</td>
<td>Sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.</td>
</tr>
<tr>
<td>BLAS_DPERMUTE (P)</td>
<td>Permutes a real (double precision) array in terms of the permutation vector P, output by DSORTV.</td>
</tr>
</tbody>
</table>
**TABLE A-13  Sort Routines (Continued)**

<table>
<thead>
<tr>
<th>Routines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS ISORT (P)</td>
<td>Sorts an integer vector X in increasing or decreasing order using quick sort algorithm.</td>
</tr>
<tr>
<td>BLAS_ISORTV (P)</td>
<td>Sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.</td>
</tr>
<tr>
<td>BLAS_IPERMUTE (P)</td>
<td>Permutes an integer array in terms of the permutation vector P, output by DSORTV.</td>
</tr>
<tr>
<td>BLAS_SSORT (P)</td>
<td>Sorts a real vector X in increasing or decreasing order using quick sort algorithm.</td>
</tr>
<tr>
<td>BLAS_SSORTV (P)</td>
<td>Sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.</td>
</tr>
<tr>
<td>BLAS_SPERMUTE (P)</td>
<td>Permutes a real array in terms of the permutation vector P, output by DSORTV.</td>
</tr>
</tbody>
</table>
Index

Symbols
%g2, %g3, %g4, and %g5 global integer registers, 22
_64, appending to routine name, 16, 27

Numerics
2D FFT routines
  complex sequences as input, 88
  conjugate symmetry, 89
  data storage format, 88
  forward 2D FFT, 88
  inverse 2D FFT, 88
  real sequences as input, 88
  routines, 78, 89
3D FFT routines
  complex sequences as input, 93
  conjugate symmetry, 93
  data storage format, 93
  forward 3D FFT, 93
  inverse 3D FFT, 93
  real sequences as input, 93
  routines, 78, 94
64-bit code
  C, 29
  Fortran 95, 28
  See also 64-bit enabled Solaris operating environment
64-bit enabled Solaris operating environment
  appending _64 to routine names, 27
  compiling code, 27
  integer promotion, 28
64-bit integer arguments, 15
  promoting integers to 64-bits, 27, 28

64-bit integer interfaces, calling, 27

A
accessible documentation, 5
architectures, 8
argument data types
  summary, 114
arguments
  convolution and correlation, 115
  FFT routines, 78
automatic code restructuring tools, 14

B
banded matrix, 37
bidirectional matrix, 136
BLAS1, 7, 151
BLAS2, 7, 152
BLAS3, 7, 153

C
C
  64-bit code, 29
  array storage, 22
  examples, 23
  routine calling conventions, 21
C interfaces
  advantages, 21
  compared to Fortran interfaces, 21
  routine calling conventions, 21
calling 64-bit integer interfaces, 27
calling conventions
  C, 21
  f77/f95, 14
CLAPACK, 8
compatibility, LAPACK, 8, 10
compile-time checking, 15
conjugate symmetric, 80
conjugate symmetry
  2D FFT routines, 89
  3D FFT routines, 93
  FFT routines, 80
convolution, 112
convolution and correlation
  arguments, 115
  routines, 114
correlation, 113
cosine transforms, 100

D
-dalign, 11, 26
data storage format
  2D FFT routines, 88
  3D FFT routines, 93
FFT routines, 81
data types
  arguments, 114
DFT
  efficiency of FFT versus DFT, 77
diagonal matrix, 136
discrete Fourier transform
  See DFT
documentation index, 4
documentation, accessing, 4 to 5

E
eempty interval, 125
enable trap 6, 12
environment variable
  STACKSIZE, 31
even sequences
  fast cosine transform routines, 101

F
-£95 interfaces
  calling conventions, 14
fast cosine transform routines, 102
  even sequences, 101
  forward and inverse, 103
  forward transform (multiple quarter-wave even
  sequences), 105
  forward transform (quarter-wave even
  sequence), 104
  inverse transform (multiple quarter-wave even
  sequences), 105
  inverse transform (quarter-wave even
  sequence), 104
  multiple sequences, 104
  quarter-wave even sequences, 101
fast Fourier transform
  See FFT
fast sine transform routines, 103
  forward and inverse, 105
  forward and inverse (multiple sequences), 106
  forward transform (multiple quarter-wave odd
  sequences), 107
  forward transform (quarter-wave odd
  sequence), 107
  inverse transform (multiple quarter-wave odd
  sequences), 107
  inverse transform (quarter-wave odd
  sequence), 106
  odd sequences, 101
  quarter-wave odd sequences, 101
features, 9
FFT
  efficiency of FFT versus DFT, 77
FFT routines
  2D FFT routines, 78
  3D FFT routines, 78
  arguments, 78
  complex sequences as input, 80
  conjugate symmetry, 80
  data storage format, 81
  forward and inverse, 78
  linear FFT routines, 78, 81
  linear forward FFT, 80
  linear forward FFT (polar form), 80
  linear inverse FFT, 80
  linear inverse FFT (polar form), 80
  real sequences as input, 80
  sequence length for most efficient
  computation, 79, 99
FFTPACK, 101, 157, 159
Fortran 95
  64-bit code, 28
  compile-time checking, 15
  optional arguments, 15, 17
  type independence, 15
  USE SUNPERF, 15
Fortran interfaces
  IBLAS routines, 127
  summary, 14

G
general band matrix, 136
general matrix, 136
general tridiagonal matrix, 138
global integer registers, 22
H
Hermitian band matrix, 139
Hermitian matrix, 139
Hermitian matrix in packed storage, 140

I
IBLAS
  add or cancel vectors routines, 129
  f95 language bindings, 128
  Fortran interfaces, 127
  matrix movements routines, 130
  matrix set operations routines, 131
  matrix utilities routines, 132
  matrix-vector operations routines, 129
  O(n^2) matrix operations routines, 130
  O(n^3) matrix operations routines, 130
  vector movements routines, 129
  vector reductions routines, 128
  vector set operations routines, 130
  vector utilities routines, 131
IBLAS naming conventions, 126
  including routines in development environment, 13
interval BLAS
  See IBLAS
  intervals, 125
    empty interval, 125

L
LAPACK, 7, 136
LAPACK 90, 8
LAPACK compatibility, 8, 10
LINPACK, 7

M
malloc, 22
man pages
  section 3P, 77, 125, 135
matrix
  banded, 37
  bidiagonal, 136
  diagonal, 136
  general, 39, 136
  general band, 136
  general tridiagonal, 138
  Hermitian, 139
  Hermitian band, 139
  Hermitian in packed storage, 140
  real orthogonal, 142
  real orthogonal in packed storage, 142
  real symmetric band, 145
  real symmetric tridiagonal, 146
  sparse, 43
  structurally symmetric sparse, 45
  symmetric, 41, 147
  symmetric in packed storage, 145
  symmetric or Hermitian-positive definite, 143
  symmetric or Hermitian-positive definite band, 143
  symmetric or Hermitian-positive definite in packed storage, 144
  symmetric or Hermitian-positive definite tridiagonal, 144
  symmetric sparse, 44
  trapezoidal, 149
  triangular, 40, 148, 149
  triangular band, 148
  triangular in packed storage, 148
  tridiagonal, 42
  unitary, 149
  unitary in packed storage, 150
  upper Hessenberg, 141
MT-safe routines, 20
multiprocessor environments, 31

N
naming conventions
  IBLAS, 126
  IBLAS prefixes, 126
Netlib, 8
Netlib Sparse BLAS
  naming conventions, 47
NIST Fortran Sparse BLAS
  naming conventions, 47
number of threads, 32

O
odd sequences
  fast sine transform routines, 101
  optional f95 arguments, 15, 17

P
packed storage, 38
parallel processing, 31
number of threads, 32
promoting integer arguments to 64-bits, 27, 28

Q
quarter-wave even sequences
  fast cosine transform routines, 101
quarter-wave odd sequences
  fast sine transform routines, 101

R
real orthogonal matrix, 142
real orthogonal matrix in packed storage, 142
real symmetric band matrix, 145
real symmetric tridiagonal matrix, 146
replacing routines, 14
routines
  2D FFT routines, 78, 89
  3D FFT routines, 78, 94
  BLAS1, 151
  BLAS2, 152
  BLAS3, 153
  C calling conventions, 21
  f95 calling conventions, 14
  fast cosine transform routines, 101, 102
  fast cosine transform routines (multiple sequences), 104
  fast sine transform routines, 101, 103
  FFTPACK, 157, 159
  forward and inverse FFT, 78
  forward fast cosine transform routines, 103
  forward fast cosine transform routines (multiple quarter-wave even sequences), 105
  forward fast cosine transform routines (quarter-wave even sequence), 104
  forward fast sine transform routines, 105
  forward fast sine transform routines (multiple quarter-wave odd sequences), 107
  forward fast sine transform routines (multiple sequences), 106
  forward fast sine transform routines (quarter-wave odd sequence), 106
  IBLAS add or cancel vectors routines, 129
  IBLAS matrix movements routines, 130
  IBLAS matrix set operations routines, 131
  IBLAS matrix utilities routines, 132
  IBLAS matrix-vector operations routines, 129
  IBLAS O(n^2) matrix operations routines, 130
  IBLAS O(n^3) matrix operations routines, 130
  IBLAS vector movements routines, 129
  IBLAS vector reductions, 128
  IBLAS vector set operations routines, 130
  IBLAS vector utilities routines, 131
  inverse fast cosine transform routines, 103
  inverse fast cosine transform routines (multiple quarter-wave even sequences), 105
  inverse fast cosine transform routines (quarter-wave even sequence), 104
  inverse fast sine transform routines, 105
  inverse fast sine transform routines (multiple quarter-wave odd sequences), 107
  inverse fast sine transform routines (multiple sequences), 106
  inverse fast sine transform routines (quarter-wave odd sequence), 106
  LAPACK, 136
  linear FFT routines, 78, 81
  sparse BLAS, 153
  VFFTPACK, 157, 159

S
section 3P man pages, 77, 125, 135
shell prompts, 4
sine transforms, 100
sparse BLAS, 153
sparse matrices
  structurally symmetric, 45
  symmetric, 44
  sparse matrix, 43
SPSOLVE sparse solver routines, 48
STACKSIZE environment variable, 31
structurally symmetric sparse matrix, 45
SuperLU sparse solver routines, 62
symmetric matrix, 41, 147
symmetric matrix in packed storage, 145
symmetric or Hermitian positive definite band matrix, 143
symmetric or Hermitian positive definite matrix, 143
symmetric or Hermitian positive definite matrix in packed storage, 144
symmetric or Hermitian positive definite tridiagonal matrix, 144
symmetric sparse matrix, 44
synchronization, 33

T
threads
  synchronization, 33
trap 6, enabling, 12
trapezoidal matrix, 149
triangular band matrix, 148
triangular matrix, 40, 148, 149
triangular matrix in packed storage, 148
tridiagonal matrix, 42
type Independence, 15
typographic conventions, 3

U
unitary matrix, 149
unitary matrix in packed storage, 150
upper Hessenberg matrix, 141
USE SUNPERF
  enabling Fortran 95 features, 15

V
VFFTPACK, 101, 157, 159

X
-xarch, 26
xFFTOPT, 100
-xlic_lib=sunperf, 11, 26
-xtypemap, 28