Sun Performance Library User's Guide

Forte Developer 7

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

Part No. 816-2463-10
May 2002, Revision A

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

Contents

Tables

Before You Begin

This book describes how to use the Sun™ specific extensions and features included with the Sun Performance Library™ subroutines that are supported by the Sun Forte Developer Fortran 95 and C compilers.

Who Should Use This Book

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

How This Book Is Organized

This book is organized into the following chapters and appendixes:

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

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

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

Chapter 4 includes information on matrix storage schemes, matrix types, and sparse matrices.

Chapter 5 describes the one-dimensional, two-dimensional, and three-dimensional fast Fourier transform routines provided with the Sun Performance Library.

Chapter 6 provides an introduction to the Interval Basic Linear Algebra Subroutine (IBLAS) library provided with the Sun Performance Library.

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

What Is Not in This Book

This book does not repeat information included in existing LAPACK books or sources on Netlib. Refer to the next section "Related Documents and Web Sites" on page xii 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*. 3rd ed., Anderson E. and others. SIAM, 1999, augments the material in this manual and provide essential information:

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

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

Online Resources

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

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

Note – LINPACK has been removed from this version of the Sun Performance Library. The LINPACK libraries and documentation are still available from `www.netlib.org`.

Typographic Conventions

TABLE P-1 Typeface Conventions

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

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

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

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*n*`]`	`O4, O`
{ }	Braces contain a set of choices for required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*dir*`[:`*dir*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*f1*`[`*,…fn*`]`	`xinline=alpha,dos`

Shell Prompts

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

Accessing Forte Developer Development Tools and Man Pages

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

For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this Forte Developer release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Forte Developer products are installed in the /opt directory. If your product software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

Accessing Forte Developer Compilers and Tools

Use the steps below to determine whether you need to change your PATH variable to access the Forte Developer compilers and tools.

▼ To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt:**

```
% echo $PATH
```

2. **Review the output for a string of paths that contain /opt/SUNWspro/bin/.**

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

▼ To Set Your PATH Environment Variable to Enable Access to Forte Developer Compilers and Tools

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

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

   /opt/SUNWspro/bin

Accessing Forte Developer Man Pages

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

▼ To Determine Whether You Need to Set Your `MANPATH` Environment Variable

1. **Request the `dbx` man page by typing the following at a command prompt:**

```
% man dbx
```

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

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

▼ To Set Your `MANPATH` Environment Variable to Enable Access to Forte Developer Man Pages

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

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

   `/opt/SUNWspro/man`

Accessing Forte Developer Documentation

You can access Forte Developer product documentation at the following locations:

- The product documentation is available from the documentation index installed with the product on your local system or network at `/opt/SUNWspro/docs/index.html`.

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

- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed product only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*

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

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be 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.

Product Documentation in Accessible Formats

Forte Developer 7 product 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 product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Release notes	Text file on the product CD at `/cdrom/devpro_v10n1_sparc/release_notes.txt`

Related Forte Developer Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html`. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

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

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

Sending Your Comments

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

`docfeedback@sun.com`

CHAPTER 1

Introduction

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

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

Libraries Included With Sun Performance Library

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

- LAPACK version 3.0 – For solving linear algebra problems.
- BLAS1 (Basic Linear Algebra Subprograms) – For performing vector-vector operations.
- BLAS2 – For performing matrix-vector operations.
- BLAS3 – For performing matrix-matrix operations.

The BLAS1, BLAS2, and BLAS3 libraries do not have version numbers. There has been only one version of the BLAS routines on Netlib.

Note – LINPACK has been removed from Sun Performance Library. LAPACK version 3.0 supersedes LINPACK and all previous versions of LAPACK. If the LINPACK routines are still needed, the LINPACK library and documentation can be obtained from www.netlib.org.

Sun Performance Library is available in both static and dynamic library versions optimized for the V8, V8+, and V9 architectures. Sun Performance Library supports static and shared libraries on Solaris 7, Solaris 8, and Solaris 9 and adds support for multiple processors.

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

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

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

Auxiliary routines are available in the shared (dynamic) libraries and the static libraries. However, there is no guarantee that auxiliary routines will continue to be available in any form in future versions of the Sun Performance Library.

Netlib

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

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

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

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

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

Sun Performance Library Features

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

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

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

Mathematical Routines

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

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

Compatibility With Previous LAPACK Versions

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

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

Getting Started With Sun Performance Library

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

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

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

or

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

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

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

To summarize, use the following:

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

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

- Specific SPARC instruction set architectures, as described in "Compiling for SPARC Platforms" on page 20.
- Parallel processing, as described in "Parallel Processing" on page 24.

Enabling Trap 6

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

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

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

2. **Assemble `trap6_handler.s`.**

```
my_system% fbe trap6_handler.s
```

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

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

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

CHAPTER 2

Using Sun Performance Library

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

Improving Application Performance

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

Replacing Routines With Sun Performance Library Routines

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

Improving Performance of Other Libraries

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

Using Tools to Restructure Code

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

Fortran 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.

Fortran SUNPERF Module for Use With Fortran 95

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

```
USE SUNPERF
```

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

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

- *Type Independence* – Sun Performance Library supports interfaces where the type of the data arguments will automatically be recognized, eliminating the need for type-dependent prefixes (S, D, C, or Z). In the FORTRAN 77 routines, the type must be specified as part of the routine name. For example, DGEMM is a double precision matrix multiply and SGEMM is a single precision matrix multiply. When calling GEMM with the Fortran 95 interfaces, Fortran will infer the type from the arguments that are passed. Passing single-precision arguments to GEMM gets results that are equivalent to specifying SGEMM, and passing double-precision arguments gets results that are equivalent to DGEMM. For example, CALL DSCAL(20,5.26D0,X,1) could be changed to CALL SCAL(20, 5.26D0, X, 1).
- *Compile-Time Checking* – In FORTRAN 77, it is generally impossible for the compiler to determine what arguments should be passed to a particular routine. In Fortran 95, the USE SUNPERF statement allows the compiler to determine the number, type, size, and shape of each argument to each Sun Performance Library routine. It can check the calls against the expected value and display errors during compilation.
- *Optional Arguments* – Sun Performance Library supports interfaces where some arguments are optional. In FORTRAN 77, all arguments must be specified in the order determined by the interface for all routines. All interfaces will support f95 style OPTIONAL attributes on arguments that are not required. Using routines with optional arguments, such as GEMM, are useful for new development. Specifically named routines, such as DGEMM, are maintained to support legacy code. To determine the optional arguments for a routine, refer to the section 3P man pages. In the section 3P man pages, optional arguments are enclosed in square brackets [ ].
- *64-bit Integer Support*– When using the 64-bit interfaces provided with Sun Performance Library, integer arguments need to be promoted to 64-bits, and the routine name needs to be modified by appending _64 to the routine name. With

the SUNPERF module, 64-bit integers will automatically be recognized, which eliminates the need for appending _64 to the routine name, as shown in the following code example.

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

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

END
```

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

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

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

END
```

For a detailed description of using the Sun Performance Library 64-bit interfaces, see "Compiling Code for a 64-Bit Enabled Solaris Operating Environment" on page 21.

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

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

Optional Arguments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The following calls to the AXPY interface are valid.

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

Fortran Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C Interfaces

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

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

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

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

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

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

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

Sun Performance Library C interfaces use the following conventions:

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

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

C Examples

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

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

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

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

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

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

CHAPTER 3

SPARC Optimization and Parallel Processing

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

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

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

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

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

Using Sun Performance Library on SPARC Platforms

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

For each -xarch option used to compile the libraries, there is a library compiled with -xparallel and a library compiled without -xparallel. When linking the program, use -dalign, -xlic_lib=sunperf, and the same command line options that were used when compiling. If -dalign cannot be used in the program, supply a trap 6 handler as described in "Getting Started With Sun Performance Library" on page 5. If compiling with a value of -xarch that is not one of [v8|v8plusa|v9a], the compiler driver will select the closest match.

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

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

Compiling for SPARC Platforms

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

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

Note – Using SPARC-specific optimization options increases application performance on the selected instruction set architecture, but limits code portability. When using these optimization options, the resulting code can be run only on systems using the specific SPARC chip from Sun Microsystems and, in some cases, a specific Solaris operating environment (32-bit or 64-bit Solaris 7, Solaris 8, or Solaris 9).

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

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

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

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

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

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

Compiling Code for a 64-Bit Enabled Solaris Operating Environment

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

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

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

64-Bit Integer Arguments

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

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

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

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

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

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

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

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

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

- To promote all default integers (integers declared without explicit byte sizes) and literal integer constants from 32 bits to 64 bits, compile with -xtypemap= integer:64.
- To promote specific integer declarations, change INTEGER or INTEGER*4 to INTEGER*8.
- To promote integer literal constants, append _8 to the constant.

Consider the following code example.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Parallel Processing

If using multithreading, use one of the following options:

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

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

or

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

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

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

or

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

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

Run-Time Issues

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

```
% setenv STACKSIZE 4000
```

Setting the `STACKSIZE` environment variable is not required for programs running with POSIX 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.

Degree of Parallelism

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Synchronization Mechanisms

The underlying parallelization model determines the Sun Performance Library behavior.

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

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

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

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

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

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

Parallel Processing Examples

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

- A single processor
- Multiple processors

Using a Single Processor

To use a single processor:

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

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

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

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

or

```
f95 -dalign -xarch=... any.f95 -xlic_lib=sunperf
```

Using Multiple Processors

To compile for multiple processors:

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

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

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

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

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

To use multiple processors:

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

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

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

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

or

```
f95 -dalign -xarch=... -xparallel any.f95 -xlic_lib=sunperf
```

CHAPTER 4

Working With Matrices

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

- Banded storage
- Packed storage

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

- General
- Triangular
- Symmetric
- Tridiagonal

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

Matrix Storage Schemes

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

Banded Storage

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

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

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

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

Packed Storage

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

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

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

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

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

Matrix Types

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

General Matrices

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

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

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

General Banded Matrix General Banded Array in Banded Storage

Triangular Matrices

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

A triangular matrix can be stored using packed storage.

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

Triangular Matrix Triangular Array in Packed Storage

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

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

Triangular Banded Matrix

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

Triangular Banded Array in Banded Storage

Symmetric Matrices

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

A symmetric matrix can be stored using packed storage.

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

Symmetric Matrix

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

Symmetric Array in Packed Storage

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

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

Symmetric Banded Matrix

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

Symmetric Banded Array in Banded Storage

Tridiagonal Matrices

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

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

Tridiagonal Matrix

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

Tridiagonal Array in Tridiagonal Storage

Sparse Matrices

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

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

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

Sparse Solver Matrix Data Formats

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

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

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

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

- Symmetric
- Structurally symmetric
- Unsymmetric

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

Symmetric Sparse Matrices

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

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

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

To represent *A* in CSC format:

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

Structurally Symmetric Sparse Matrices

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

An example of a structurally symmetric matrix is shown below.

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

To represent *A* in CSC format:

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

Unsymmetric Sparse Matrices

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

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

To represent A in CSC format:

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

Sun Performance Library Sparse BLAS

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

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

Refer to the following sources for additional sparse BLAS information.

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

Naming Conventions

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

Netlib Sparse BLAS

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

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

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

TABLE 4-1 Netlib Sparse BLAS Naming Conventions

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

The prefix can be one of the following data types:

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

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

NIST Fortran Sparse BLAS

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

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

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

TABLE 4-2 NIST Fortran Sparse BLAS Routine Naming Conventions

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

Sparse Solver Routines

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

TABLE 4-3 Sparse Solver Routines

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

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

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

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

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

Routine Calling Order

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

TABLE 4-4 Sparse Solver Routine Calling Order

One Call Interface: For solving single matrix	
Start	
`DGSSFS()`	Initialize, order, factor, solve
`DGSSSL()`	Additional solves (optional): repeat `dgsssl()` as needed
`DGSSDA()`	Deallocate working storage
Finish	
End of One-Call Interface	
Regular Interface: For solving multiple matrices with the same structure	
Start	
`DGSSIN()`	Initialize
`DGSSOR()`	Order
`DGSSFA()`	Factor
`DGSSSL()`	Solve: repeat `dgssfa()` or `dgsssl()` as needed
`DGSSDA()`	Deallocate working storage
Finish	
End of Regular Interface	

Sparse Solver Examples

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

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

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

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

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

```
      ldrhs  = 5
      outunt = 6
      msglvl = 0
      ordmthd = 'mmd'
c
c  call single call interface
c
      call dgssfs ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              values, nrhs  , rhs,    ldrhs , ordmthd,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

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

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

      end
```

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

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

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

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

      implicit none

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

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

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

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

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

  200 format(a5,3a20)

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

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

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

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

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

      integer           neqns, ier, msglvl, outunt, ldrhs, nrhs
      character         mtxtyp*2, pivot*1, ordmthd*3
      double precision  handle(150)
      integer           colstr(5), rowind(8)
      double precision  values(8), rhs(4), xexpct(4)
      integer           i
```

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

```
c
c  Sparse matrix structure and value arrays.  Coefficient matrix
c    has a symmetric structure and unsymmetric values.
c    Ax = b, (solve for x) where:
c
c      1.0   3.0   0.0   0.0       1.0        7.0
c      2.0   4.0   0.0   7.0       2.0       38.0
c  A = 0.0   0.0   6.0   0.0   x = 3.0   b = 18.0
c      0.0   5.0   0.0   8.0       4.0       42.0
c
      data colstr / 1, 3, 6, 7, 9 /
      data rowind / 1, 2, 1, 2, 4, 3, 2, 4 /
      data values / 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0, 6.0d0, 7.0d0,
     &              8.0d0 /
      data rhs    / 7.0d0, 38.0d0, 18.0d0, 42.0d0 /
      data xexpct / 1.0d0, 2.0d0, 3.0d0, 4.0d0 /
c
c  initialize solver
c
      mtxtyp= 'su'
      pivot = 'n'
      neqns  = 4
      outunt = 6
      msglvl = 0
c
c  call regular interface
c
      call dgssin ( mtxtyp, pivot,  neqns , colstr, rowind,
     &              outunt, msglvl, handle, ier             )
      if ( ier .ne. 0 ) goto 110
c
c  ordering and symbolic factorization
c
      ordmthd = 'mmd'
      call dgssor ( ordmthd, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  numeric factorization
c
      call dgssfa ( neqns, colstr, rowind, values, handle, ier )
      if ( ier .ne. 0 ) goto 110
```

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

```
c
c  solution
c
      nrhs   = 1
      ldrhs  = 4
      call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

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

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

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

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

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

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

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

```
c
c  ordering and symbolic factorization
c
      ordmthd = 'mmd'
      call dgssor ( ordmthd, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  numeric factorization
c
      call dgssfa ( neqns, colstr, rowind, values, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  solution
c
      nrhs   = 1
      ldrhs  = 5
      call dgsssl ( nrhs, rhs, ldrhs, handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier )
      if ( ier .ne. 0 ) goto 110
c
c  print values of sol
c
      write(6,200) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
      do i = 1, neqns
        write(6,300) i, rhs(i), xexpct(i), (rhs(i)-xexpct(i))
      enddo
      stop
  110 continue
c
c call to sparse solver returns an error
c
      write ( 6 , 400 )
     &      ' example: FAILED sparse solver error number = ', ier
      stop

  200 format(a5,3a20)

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

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

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

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

References

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

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

CHAPTER 5

Using Sun Performance Library 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 `SFFTC` routine, type **`man -s 3P sfftc`**. Routine names must be lowercase. For an overview of the FFT routines, type **`man -s 3P fft`**.

Forward and Inverse FFT Routines

TABLE 5-1 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 5-1 FFT Routines and Their Arguments

Routine Name	Arguments
Linear Routines	
`CFFTS [ZFFTD]`	`(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)`
`SFFTC [DFFTZ]`	`(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)`
`CFFTSM [ZFFTDM]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
`SFFTCM [DFFTZM]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
`CFFTC [ZFFTZ]`	`(OPT, N1, SCALE, X, Y, TRIGS, IFAC, WORK, LWORK, ERR)`
`CFFTCM [ZFFTZM]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
Two-Dimensional Routines	
`CFFTS2 [ZFFTD2]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
`SFFTC2 [DFFTZ2]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
`CFFTC2 [ZFFTZ2]`	`(OPT, N1, N2, SCALE, X, LDX1, Y, LDY1, TRIGS, IFAC, WORK, LWORK, ERR)`
Three-Dimensional Routines	
`CFFTS3 [ZFFTD3]`	`(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)`
`SFFTC3 [DFFTZ3]`	`(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)`
`CFFTC3 [ZFFTZ3]`	`(OPT, N1, N2, N3, SCALE, X, LDX1, LDX2, Y, LDY1, LDY2, TRIGS, IFAC, WORK, LWORK, ERR)`

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 tranform. `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 tranform. `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 tranforms, 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 acommodate the transform results when computing an in-place tranform.
- `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.

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

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

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

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

The inverse FFT routines compute

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

or in polar form,

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

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,

$$X(k) = X^*(N1 - k), \qquad k = \frac{N1}{2} + 1, \ldots, N1 - 1 \ .$$

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 5-2 lists the single precision linear FFT routines and their purposes. For routines that have two-dimensional arrays as input and output, TABLE 5-2 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 5-2 for the mapping. See the individual man pages for a complete description of the routines and their arguments.

TABLE 5-2 Single Precision Linear FFT Routines

Name	Purpose	Size and Type of Input	Size and Type of Output	Leading Dimension Requirements	
				In-place	Out-of-Place
`SFFTC`	`OPT` = 0 initialization				
	`OPT` = -1 real-to-complex forward linear FFT of a single vector	$N1$, Real	$\frac{N1}{2}+1$, Complex		
`SFFTC`	`OPT` = 0 initialization				
	`OPT` = 1 complex-to-real inverse linear FFT of single vector	$\frac{N1}{2}+1$, Complex	$N1$ Real		
`CFFTC`	`OPT` = 0 initialization				
	`OPT` = -1 complex-to-complex forward linear FFT of a single vector	$N1$, Complex	$N1$, Complex		
	`OPT` = 1 complex-to-complex inverse linear FFT of a single vector	$N1$, Complex	$N1$, Complex		
`SFFTCM`	`OPT` = 0 initialization				
	`OPT` = -1 real-to-complex forward linear FFT of M vectors	$N1 \times M$, Real	$\left(\frac{N1}{2}+1\right) \times M$, Complex	`LDX1` $= 2 \times$ `LDY1`	`LDX1` $\geq N1$
`CFFTSM`	`OPT` = 0 initialization				
	`OPT` = 1 complex-to-real inverse linear FFT of M vectors	$\left(\frac{N1}{2}+1\right) \times M$, Complex	$N1 \times M$, Real	`LDX1` $\geq \frac{N1}{2}+1$ `LDY1`$=2 \times$ `LDX1`	`LDX1` $\geq \frac{N1}{2}+1$ `LDY1` $\geq N1$
`CFFTCM`	`OPT` = 0 initialization				
	`OPT` = -1 complex-to-complex forward linear FFT of M vectors	$N1 \times M$, Complex	$N1 \times M$, Complex	`LDX1` $\geq N1$ `LDY1` $\geq N1$	`LDX1` $\geq N1$ `LDY1` $\geq N1$
	`OPT` = 1 complex-to-complex inverse linear FFT of M vectors	$N1 \times M$, Complex	$N1 \times M$, Complex	`LDX1` $\geq N1$ `LDY1` $\geq N1$	`LDX1` $\geq N1$ `LDY1` $\geq N1$

TABLE 5-2 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 *N*1 < 0

CODE EXAMPLE 5-1 shows how to compute the linear real-to-complex and complex-to-real FFT of a set of sequences.

CODE EXAMPLE 5-1 Linear Real-to-Complex FFT and Complex-to-Real FFT

```
my_system% cat testscm.f
       PROGRAM TESTSCM
       IMPLICIT NONE
       INTEGER :: LW, IERR, I, J, K, LDX, LDC
       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 = (/.1, .2, .3,0.0,0.0,0.0,7.,8.,9.,
      $    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(*,*)
* intialize 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
```

CODE EXAMPLE 5-1 Linear Real-to-Complex FFT and Complex-to-Real FFT *(Continued)*

```
* 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, 0, 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.
* X must be large enough to store N1/2+1 complex values
      CALL SFFTCM(-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 CFFTSM(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 CFFTSM(1, N1, N2, SCALE, Z, LDZ, Z, LDZ*2, TRIGS,
     $            IFAC, SW, 0, IERR)
      IF (IERR .NE. 0) THEN
         PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
         STOP
      END IF
```

CODE EXAMPLE 5-1 Linear Real-to-Complex FFT and Complex-to-Real FFT *(Continued)*

```
      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
      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, F4.1,A1,F4.1,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 testscm.f -xlic_lib=sunperf
my_system% a.out
Linear complex-to-real and real-to-complex FFT of a sequence
X =
0.1 7.0
0.2 8.0
0.3 9.0
out-of-place forward FFT of X:
Z =
( 0.6, 0.0) (24.0, 0.0)
(-0.2, 0.1) (-1.5, 0.9)
in-place forward FFT of X:
( 0.6, 0.0) (24.0, 0.0)
(-0.2, 0.1) (-1.5, 0.9)
out-of-place inverse FFT of Z:
0.1 7.0
0.2 8.0
0.3 9.0
```

CODE EXAMPLE 5-1 Linear Real-to-Complex FFT and Complex-to-Real FFT *(Continued)*

```
in-place inverse FFT of Z:
0.1 7.0
0.2 8.0
0.3 9.0
```

CODE EXAMPLE 5-1 Notes:

The forward FFT of X is actually

	(0.6, 0.0)	(24.0, 0.0)
Z =	(-0.2, 0.1)	(-1.5, 0.9)
	(-0.2, -0.1)	(-1.5, -0.9)

Because of symmetry, Z(2) is the complex conjugate of Z(1), and therefore only the first two $\frac{N1}{2} + 1 = 2$ 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 × 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 × 2.

CODE EXAMPLE 5-2 shows how to compute the linear complex-to-complex FFT of a set of sequences.

CODE EXAMPLE 5-2 Linear Complex-to-Complex FFT

```
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)
```

CODE EXAMPLE 5-2 Linear Complex-to-Complex FFT *(Continued)*

```
       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 WRITE(*,*)
* 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
         PRINT*,'ROUTINE RETURN WITH ERROR CODE = ', IERR
         STOP
       END IF
* Compute out-of-place forward linear FFT.
* Let FFT routine allocate memory.
       CALL CFFTCM(-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 CFFTCM(-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
```

CODE EXAMPLE 5-2 Linear Complex-to-Complex FFT *(Continued)*

```
       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 CFFTCM(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
X =
( 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)
in-place forward FFT of X:
(  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)
out-of-place forward FFT of X:
Y =
(  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)
in-place inverse FFT of Y:
Y =
( 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)
```

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

$$X(k, n) = \sum_{l=0}^{N2-1} \sum_{j=0}^{N1-1} x(j, l) e^{\frac{-2\pi i l n}{N2}} e^{\frac{-2\pi i j k}{N1}}, \qquad k = 0, \ldots, N1-1, n = 0, \ldots, N2-1 ,$$

and the inverse two-dimensional FFT routines compute

$$x(j, l) = \sum_{n=0}^{N2-1} \sum_{k=0}^{N1-1} X(k, n) e^{\frac{2\pi i l n}{N2}} e^{\frac{2\pi i j k}{N1}}, \qquad j = 0, \ldots, N1-1, l = 0, \ldots, N2-1.$$

For both the forward and inverse two-dimensional transforms, a complex-to-complex transform where the input problem is $N1 \times N2$ will yield a complex array that is also $N1 \times N2$.

When computing a real-to-complex two-dimensional transform (forward FFT), if the real input array is of dimensions $N1 \times N2$, the result will be a complex array of dimensions $(\frac{N1}{2} + 1) \times N2$. Conversely, when computing a complex-to-real transform (inverse FFT) of dimensions N1 × N2, an $(\frac{N1}{2} + 1) \times N2$ 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 $\frac{N1}{2} + 1$ complex data points need to be stored in the input or output array along the first dimension. The complex subarray $X(\frac{N1}{2} + 1{:}N1 - 1, :)$ can be obtained from $X(0{:}\frac{N1}{2}, :)$ as follows:

$$\begin{aligned} X(k, n) &= X^{*}(N1 - k, n), \\ k &= \frac{N1}{2} + 1, \ldots, N1 - 1 \\ n &= 0, \ldots, N2 - 1 \end{aligned}$$

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 $N1$ and $N2$ and the trigonometric weights associated with those factors. In subsequent forward or inverse transforms, initialization is not necessary as long as $N1$ and $N2$ 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, $N = N1$ in the complex-to-complex and complex-to-real transforms and $N = \frac{N1}{2} + 1$ in the real-to-complex transform.

TABLE 5-3 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 5-3 for the mapping. Refer to the individual man pages for a complete description of the routines and their arguments.

TABLE 5-3 Single Precision Two-Dimensional FFT Routines

Name	Purpose	Size, Type of Input	Size, Type of Output	Leading Dimension Requirements	
				In-place	Out-of-Place
`SFFTC2`	`OPT` = 0 initialization				
	`OPT` = -1 real-to-complex forward two-dimensional FFT	N1 × N2, Real	$(\frac{N1}{2}+1)\times N2$, Complex	`LDX1` = 2 × `LDY1` `LDY1` ≥ $\frac{N1}{2}+1$	`LDX1` ≥ $N1$ `LDY1` ≥ $\frac{N1}{2}+1$
`CFFTS2`	`OPT` = 0 initialization				
	`OPT` = 1 complex-to-real inverse two-dimensional FFT	$(\frac{N1}{2}+1)\times N2$, Complex	$N1 \times N2$, Real	`LDX1` ≥ $\frac{N1}{2}+1$ `LDY1`=2 × `LDX1`	`LDX1` ≥ $\frac{N1}{2}+1$ `LDY1`≥ 2 × `LDX1` `LDY1` is even
`CFFTC2`	`OPT` = 0 initialization				
	`OPT` = -1 complex-to-complex forward two-dimensional FFT	$N1 \times N2$, Complex	$N1 \times N2$, Complex	`LDX1` ≥ $N1$ `LDY1` = `LDX1`	`LDX1` ≥ $N1$ `LDY1` ≥ $N1$
	`OPT` = 1 complex-to-complex inverse two-dimensional FFT	$N1 \times N2$, Complex	$N1 \times N2$, Complex	`LDX1` ≥ $N1$ `LDY1` = `LDX1`	`LDX1` ≥ $N1$ `LDY1` = `LDX1`

TABLE 5-3 Notes:

- `LDX1` is leading dimension of input array.
- `LDY1` is leading dimension of output array.
- `N1` is first dimension of the FFT problem.
- `N2` is 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 5-3 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 SFFTC2(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 SFFTC2(-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(*,*)
```

CODE EXAMPLE 5-3 Two-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Two-Dimensional Array *(Continued)*

```
* 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 SFFTC2(-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)
      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
```

CODE EXAMPLE 5-3 Two-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Two-Dimensional Array *(Continued)*

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

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

$$X(k,n,m) = \sum_{h=0}^{N3-1} \sum_{l=0}^{N2-1} \sum_{j=0}^{N1-1} x(j,l,h)\ e^{\frac{-2\pi ihm}{N3}} e^{\frac{-2\pi iln}{N2}} e^{\frac{-2\pi ijk}{N1}},$$

$k = 0, \ldots, N1 - 1$

$n = 0, \ldots, N2 - 1$

$m = 0, \ldots, N3 - 1$

and the inverse FFT computes

$$x(j,l,h) = \sum_{m=0}^{N3-1} \sum_{n=0}^{N2-1} \sum_{k=0}^{N1-1} X(k,n,m)\ e^{\frac{2\pi ihm}{N3}} e^{\frac{2\pi iln}{N2}} e^{\frac{2\pi ijk}{N1}},$$

$j = 0, \ldots, N1 - 1$

$l = 0, \ldots, N2 - 1$

$h = 0, \ldots, N3 - 1$

In the complex-to-complex transform, if the input problem is $N1 \times N2 \times N3$, a three-dimensional transform will yield a complex array that is also $N1 \times N2 \times N3$. When computing a real-to-complex three-dimensional transform, if the real input array is of dimensions $N1 \times N2 \times N3$, the result will be a complex array of dimensions $(\frac{N1}{2} + 1) \times N2 \times N3$. Conversely, when computing a complex-to-real FFT of dimensions $N1 \times N2 \times N3$, an $(\frac{N1}{2} + 1) \times N2 \times N3$ 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 $\frac{N1}{2} + 1$ complex data points need to be stored along the first dimension. The complex subarray $X(\frac{N1}{2} + 1{:}N1 - 1,\ :, :)$ can be obtained from $X(0{:}\frac{N1}{2}, :, :)$ as follows:

$$\begin{aligned} X(k,n,m) &= X^*(N1-k,n,m), \\ k &= \frac{N1}{2} + 1, \ldots N1 - 1 \\ n &= 0, \ldots, N2 - 1 \\ m &= 0, \ldots, N3 - 1 \end{aligned}$$

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 $N1$, $N2$, and $N3$ and the trigonometric weights associated with those factors. In subsequent forward or inverse transforms, initialization is not necessary as long as $N1$, $N2$, and $N3$ remain unchanged.

IMPORTANT: Upon returning from a three-dimensional FFT routine, $Y(0 : N - 1, :, :)$ contains the transform results and the original contents of $Y(N$:`LDY1`-1, :, :) is overwritten. Here, $N = N1$ in the complex-to-complex and complex-to-real transforms and $N = \frac{N1}{2} + 1$ in the real-to-complex transform.

TABLE 5-4 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 5-4 for the mapping. See the individual man pages for a complete description of the routines and their arguments.

TABLE 5-4 Single Precision Three-Dimensional FFT Routines

Name	Purpose	Size, Type of Input	Size, Type of Output	Leading Dimension Requirements	
				In-place	Out-of-Place
`SFFTC3`	`OPT` = 0 initialization				

	`OPT` = -1 real-to-complex forward three-dimensional FFT	$N1 \times N2 \times N3$, Real	$(\frac{N1}{2} + 1) \times N2 \times N3$, Complex	`LDX1`=$2 \times$ `LDY1`
				`LDX2` $\geq N2$
				`LDY1` $\geq \frac{N1}{2} + 1$
				`LDY2` = `LDX2`
					`LDX1` $\geq N1$
					`LDX2` $\geq N2$
					`LDY1` $\geq \frac{N1}{2} + 1$
					`LDY2` $\geq N2$

`CFFTS3`	`OPT` = 0 initialization				

	`OPT` = 1 complex-to-real inverse three-dimensional FFT	$(\frac{N1}{2} + 1) \times N2 \times N3$, Complex	$N1 \times N2 \times N3$, Real	`LDX1` $\geq \frac{N1}{2} + 1$
				`LDX2` $\geq N2$
				`LDY1`=$2 \times$ `LDX1`
				`LDY2`=`LDX2`
					`LDX1` $\geq \frac{N1}{2} + 1$
					`LDX2` $\geq N2$
					`LDY1` $\geq 2 \times$ `LDX1`
					`LDY1` is even
					`LDY2` $\geq N2$

TABLE 5-4 Single Precision Three-Dimensional FFT Routines

Name	Purpose	Size, Type of Input	Size, Type of Output	Leading Dimension Requirements	
				In-place	Out-of-Place
`CFFTC3`	`OPT` = 0 initialization				

	`OPT` = -1 complex-to-complex forward three-dimensional FFT	$N1 \times N2 \times N3$, Complex	$N1 \times N2 \times N3$, Complex	`LDX1` ≥ $N1$
				`LDX2` ≥ $N2$
				`LDY1=LDX1`
				`LDY2=LDX2`
					`LDX1` ≥ $N1$
					`LDX2` ≥ $N2$
					`LDY1` ≥ $N1$
					`LDY2` ≥ $N2$

	`OPT` = 1 complex-to-complex inverse three-dimensional FFT	$N1 \times N2 \times N3$, Complex	$N1 \times N2 \times N3$, Complex	`LDX1` ≥ $N1$
				`LDX2` ≥ $N2$
				`LDY1=LDX1`
				`LDY2=LDX2`
					`LDX1` ≥ $N1$
					`LDX2` ≥ $N2$
					`LDY1` ≥ $N1$
					`LDY2` ≥ $N2$

TABLE 5-4 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 5-4 shows how to compute the three-dimensional real-to-complex FFT and complex-to-real FFT of a three-dimensional array.

CODE EXAMPLE 5-4 Three-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Three-Dimensional Array

```
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)
```

CODE EXAMPLE 5-4 Three-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Three-Dimensional Array *(Continued)*

```
       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.,1.0,1.1,1.2,0.,
      $     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
          WRITE(*,*)
       END DO
* 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
          WRITE(*,*)
       END DO
```

CODE EXAMPLE 5-4 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 SFFTC3(-1,N1,N2,N3,ONE,V,LDR1,LDR2,V,LDR1/2,LDR2,TRIGS,
     $            IFAC,SW,LW,IERR)
      WRITE(*,*) '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 CFFTS3(1,N1,N2,N3,SCALE,Y,LDY1,LDY2,Z,LDR1,LDR2,TRIGS,
     $            IFAC,SW,0,IERR)
      WRITE(*,*) '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
        WRITE(*,*)
      END DO
* Compute 3-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 CFFTS3(1,N1,N2,N3,SCALE,Y,LDY1,LDY2,Y,2*LDY1,LDY2,
     $            TRIGS,IFAC,SW,LW,IERR)
      WRITE(*,*) 'in-place inverse FFT of Y:'
      CALL PRINT_COMPLEX_AS_REAL(N1, N2, N3, Y, 2*LDY1, LDY2)
      DEALLOCATE(SW)
      END PROGRAM TESTSC3
      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, *)
```

CODE EXAMPLE 5-4 Three-Dimensional Real-to-Complex FFT and Complex-to-Real FFT of a Three-Dimensional Array *(Continued)*

```
       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 testsc3.f -xlic_lib=sunperf**
my_system% **a.out**
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
```

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 5-2, TABLE 5-3, and TABLE 5-4.

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*1 real data points correspond to $\frac{N1}{2} + 1$ complex data points. *N*1 real data points do map to *N*1 complex data points, but because there is conjugate symmetry in the complex data, only $\frac{N1}{2} + 1$ 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

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

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 $p, q, r, s, t, u, v \geq 0$.

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

CODE EXAMPLE 5-5 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
```

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/`).

Fast Cosine and Sine Transform Routines

TABLE 5-5 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 5-5 Fast Cosine and Sine Transform Routines and Their Arguments

Name	Arguments
Fast Cosine Transforms for Even Sequences	
`COST [DCOST]`	`(LEN+1, X, WORK)`
`COSTI [DCOSTI]`	`(LEN+1, WORK)`
`VCOST [VDCOST]`	`(M, LEN+1, X, WORK, LD, TABLE)`
`VCOSTI [VDCOSTI]`	`(LEN+1, TABLE)`
Fast Cosine Transforms for Quarter-Wave Even Sequences	
`COSQF [DCOSQF]`	`(LEN, X, WORK)`
`COSQB [DCOSQB]`	`(LEN, X, WORK)`
`COSQI [DCOSQI]`	`(LEN, WORK)`
`VCOSQF [VDCOSQF]`	`(M, LEN, X, WORK, LD, TABLE)`
`VCOSQB [VDCOSQB]`	`(M, LEN, X, WORK, LD, TABLE)`
`VCOSQI [VDCOSQI]`	`(LEN, TABLE)`

TABLE 5-5 Fast Cosine and Sine Transform Routines and Their Arguments

Name	Arguments
Fast Sine Transforms for Odd Sequences	
`SINT [DSINT]`	`(LEN-1, X, WORK)`
`SINTI [DSINTI]`	`(LEN-1, WORK)`
`VSINT [VDSINT]`	`(M, LEN-1, X, WORK, LD, TABLE)`
`VSINTI [VDSINTI]`	`(LEN-1, TABLE)`
Fast Sine Transforms for Quarter-Wave Odd Sequences	
`SINQF [DSINQF]`	`(LEN, X, WORK)`
`SINQB [DSINQB]`	`(LEN, X, WORK)`
`SINQI [DSINQI]`	`(LEN, WORK)`
`VSINQF [VDSINQF]`	`(M, LEN, X, WORK, LD, TABLE)`
`VSINQB [VDSINQB]`	`(M, LEN, X, WORK, LD, TABLE)`
`VSINQI [VDSINQI]`	`(LEN, TABLE)`

TABLE 5-5 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.

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 $n = -N + 1, \dots, 0, \dots, N$. 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 $n = -N + 1, \dots, 0, \dots, N$ 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 unormalized, and if scaled by $\frac{1}{4N}$, 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.

Fast Sine Transforms

Another type of symmetry that is commonly encountered is the odd symmetry where $x(n) = -x(-n)$ for $n = -N+1, \dots, 0, \dots, N$. 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 SINT computes the FST of a single real odd sequence while VSINT computes the FST of one or more sequences. Before calling [V]SINT, [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 VSINT twice will result in the original N -1 data points. Calling SINT twice will result in the original N -1 data points multiplied by $2N$.

An odd sequence with symmetry such that $x(n) = -x(-n - 1)$, where $n = -N + 1, \dots, 0, \dots,$ N is said to have quarter-wave odd symmetry. SINQF and SINQB compute the FST and its inverse, respectively, of a single real quarter-wave odd sequence while VSINQF and VSINQB operate on one or more sequences. SINQB is unnormalized, so using the results of SINQF as input in SINQB produces the original sequence scaled by a factor of $4N$. However, VSINQB is normalized, so a call to VSINQF followed by a call to 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 [V]SINQI.

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.

`[D]COST`: Forward and Inverse Fast Cosine Transform (FCT) of a Sequence

The forward and inverse FCT of a sequence is computed as

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

`[D]COST` Notes:

- $N + 1$ values are needed to compute the FCT of an N-point sequence.
- `[D]COST` also computes the inverse transform. When `[D]COST` is called twice, the result will be the original sequence scaled by $\frac{1}{2N}$.

`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 $i = 0, M - 1$

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

`V[D]COST` *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.

`[D]COSQF`: Forward FCT of a Quarter-Wave Even Sequence

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

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

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

`[D]COSQB`: Inverse FCT of a Quarter-Wave Even Sequence

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

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

Calling the forward and inverse routines will result in the original input scaled by $\frac{1}{4N}$.

`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 $i = 0, M - 1$

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

`V[D]COSQF` Notes:

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

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

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

V[D]COSQB Notes:

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

[D]SINT: Forward and Inverse Fast Sine Transform (FST) of a Sequence

The forward and inverse FST of a sequence is computed as

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

[D]SINT Notes:

- N-1 values are needed to compute the FST of an N-point sequence.
- [D]SINT also computes the inverse transform. When [D]SINT is called twice, the result will be the original sequence scaled by $\frac{1}{2N}$.

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 $i = 0, M - 1$

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

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.

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

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

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

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

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

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

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

Calling the forward and inverse routines will result in the original input scaled by $\frac{1}{4N}$.

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

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

V[D]SINQF Notes:

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

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

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

V[D]SINQB Notes:

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

Fast Cosine Transform Examples

CODE EXAMPLE 5-6 calls `COST` to compute the FCT and the inverse transform of a real even sequence. If the real sequence is of length 2*N*, 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 5-6 Compute FCT and Inverse FCT of Single Real Even Sequence

```
my_system% cat cost.f
       program 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(*,*)
    $   'Inverse fast cosine transform (results scaled by 1/2*N)'
       write(*,'(5(f8.3,2x),/)')(z(i)*scale,i=0,len)
       end
my_system% f95 -dalign 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 5-7 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 2*N*, 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 5-7 Compute the FCT and the Inverse FCT of Two Real Quarter-wave Even Sequences

```
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
```

CODE EXAMPLE 5-7 Compute the FCT and the Inverse FCT of Two Real Quarter-wave Even Sequences

```
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 )
```

Fast Sine Transform Examples

In CODE EXAMPLE 5-8, `SINT` is called to compute the FST and the inverse transform of a real odd sequence. If the real sequence is of length 2*N*, 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 5-8 Compute FST and the Inverse FST of a Real Odd Sequence

```
my_system% cat sint.f
      program 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)
```

CODE EXAMPLE 5-8 Compute FST and the Inverse FST of a Real Odd Sequence *(Continued)*

```
      call sint(len-1, z, work)
      write(*,*) $ 'Inverse fast sine transform (results scaled by 1/2*N)'
      write(*,'(3(f8.3,2x),/)')(z(i)*scale,i=0,len-2)
      end
my_system% f95 -dalign sint.f -xlic_lib=sunperf
my_system% a.out
Input sequence of length 4 requires 3 data points
0.557 0.603 0.210
Forward fast sine transform
2.291 0.694 -0.122
Inverse fast sine transform (results scaled by 1/2*N)
0.557 0.603 0.210
```

In CODE EXAMPLE 5-9 `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 2*N*, 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 5-9 Compute FST and Inverse FST of Two Real Quarter-Wave Odd Sequences

```
my_system% cat vsinq.f
      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(*,*)
    $ '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
```

CODE EXAMPLE 5-9 Compute FST and Inverse FST of Two Real Quarter-Wave Odd Sequences

```
      call vsinqb(m,len, z, xt, ld, work)
      write(*,*)
    $ 'Inverse fast sine transform for quarter-wave odd 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
my_system% f95 vsinq.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 sine transform for quarter-wave odd sequences
seq1 = (0.823 0.057 0.078 0.305 )
seq2 = (0.654 0.466 -.069 -.037 )
Inverse fast sine transform for quarter-wave odd sequences
(results are normalized)
seq1 = (0.557 0.352 0.990 0.539 )
seq2 = (0.603 0.867 0.417 0.156 )
```

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.

Convolution

Given two functions $x(t)$ and $y(t)$, the Fourier transform of the convolution of $x(t)$ and $y(t)$, denoted as $x \star y$, is the product of their individual Fourier transforms: $\mathrm{DFT}(x \star y) = X \odot Y$ where $\star$ 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, $j = 0, \ldots, N - 1$, 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

$$(x \star y)_j \equiv \sum_{k=\frac{-N}{2}+1}^{\frac{N}{2}} x_{j-k} y_k, \qquad j = 0, \ldots, N-1 .$$

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 $Z = X \odot Y \Leftrightarrow$ DFT$(x \star y)$
- Compute z = inverse FFT of Z; $z = (x \star y)$

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.

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:

$$\mathrm{Corr}(x, y)_j \equiv \sum_{k=0}^{N-1} x_{j+k} y_k, \qquad j = 0, \ldots, N-1 \ .$$

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.

Sun Performance Library Convolution and Correlation Routines

Sun Performance Library contains the convolution routines shown in TABLE 5-6.

TABLE 5-6 Convolution and Correlation Routines

Routine	Arguments	Function
`SCNVCOR`, `DCNVCOR`, `CCNVCOR`, `ZCNVCOR`	`CNVCOR,FOUR,NX,X,IFX, INCX,NY,NPRE,M,Y,IFY, INC1Y,INC2Y,NZ,K,Z, IFZ,INC1Z,INC2Z,WORK, LWORK`	Convolution or correlation of a filter with one or more vectors
`SCNVCOR2`, `DCNVCOR2`, `CCNVCOR2`, `ZCNVCOR2`	`CNVCOR,METHOD,TRANSX, SCRATCHX,TRANSY, SCRATCHY,MX,NX,X,LDX, MY,NY,MPRE,NPRE,Y,LDY, MZ,NZ,Z,LDZ,WORKIN, LWORK`	Two-dimensional convolution or correlation of two matrices
`SWIENER`, `DWIENER`	`N_POINTS,ACOR,XCOR, FLTR,EROP,ISW,IERR`	Wiener deconvolution of two signals

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

Arguments for Convolution and Correlation Routines

The one-dimensional convolution and correlation routines use the arguments shown in TABLE 5-7.

TABLE 5-7 Arguments for One-Dimensional Convolution and Correlation Routines SCNVCOR, DCNVCOR, CCNVCOR, and ZCNVCOR

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

TABLE 5-7 Arguments for One-Dimensional Convolution and Correlation Routines SCNVCOR, DCNVCOR, CCNVCOR, and ZCNVCOR *(Continued)*

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

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

The two-dimensional convolution and correlation routines use the arguments shown in TABLE 5-8.

TABLE 5-8 Arguments for Two-Dimensional Convolution and Correlation Routines SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2

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

TABLE 5-8 Arguments for Two-Dimensional Convolution and Correlation Routines SCNVCOR2, DCNVCOR2, CCNVCOR2, and ZCNVCOR2 *(Continued)*

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

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

Work Array WORK for Convolution and Correlation Routines

The minimum dimensions for the WORK work arrays used with the one-dimensional and two-dimensional convolution and correlation routines are shown in TABLE 5-11. 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 5-9.

TABLE 5-9 Arguments Affecting Minimum Work Array Size for Two-Dimensional Routines: `SCNVCOR2`, `DCNVCOR2`, `CCNVCOR2`, and `ZCNVCOR2`

Argument	Definition
`MX`	Number of rows in the filter matrix
`MY`	Number of rows in the input matrix
`MZ`	Number of output vectors
`NX`	Number of columns in the filter matrix
`NY`	Number of columns in the input matrix
`NZ`	Length of output vectors
`MPRE`	Number of implicit zeros prefixed to each row of the input matrix
`NPRE`	Number of implicit zeros prefixed to each column of the input matrix
`MPOST`	`MAX(0,MZ-MYC)`
`NPOST`	`MAX(0,NZ-NYC)`
`MYC`	`MPRE` + `MPOST` + `MYC_INIT`, where `MYC_INIT` depends upon filter and input matrices, as shown in TABLE 5-10
`NYC`	`NPRE` + `NPOST` + `NYC_INIT`, where `NYC_INIT` depends upon filter and input matrices, as shown in TABLE 5-10

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

TABLE 5-10 `MYC_INIT` and `NYC_INIT` Dependencies

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

The values assigned to the minimum work array size is shown in TABLE 5-11.

TABLE 5-11 Minimum Dimensions and Data Types for `WORK` Work Array Used With Convolution and Correlation Routines

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

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

Sample Program: Convolution

CODE EXAMPLE 5-10 uses `CCNVCOR` to perform FFT convolution of two complex vectors.

CODE EXAMPLE 5-10 One-Dimensional Convolution Using Fourier Transform Method and `COMPLEX` Data

```
my_system% cat con_ex20.f
      PROGRAM TEST
C
      INTEGER           LWORK
      INTEGER           N
      PARAMETER        (N = 3)
      PARAMETER        (LWORK = 4 * N + 15)
      COMPLEX           P1(N), P2(N), P3(2*N-1), WORK(LWORK)
      DATA P1 / 1, 2, 3 /,  P2 / 4, 5, 6 /
C
      EXTERNAL          CCNVCOR
C
      PRINT *, 'P1:'
      PRINT 1000, P1
      PRINT *, 'P2:'
      PRINT 1000, P2
      CALL CCNVCOR ('V', 'T', N, P1, 1, 1, N, 0, 1, P2, 1, 1, 1,
```

CODE EXAMPLE 5-10 One-Dimensional Convolution Using Fourier Transform Method and `COMPLEX` Data *(Continued)*

```
     $              2 * N - 1, 1, P3, 1, 1, 1, WORK, LWORK)
C
      PRINT *, 'P3:'
      PRINT 1000, P3
C
 1000 FORMAT (1X, 100(F4.1,' +',F4.1,'i  '))
C
      END
my_system% f95 -dalign con_ex20.f -xlic_lib=sunperf
my_system% a.out
 P1:
  1.0 + 0.0i   2.0 + 0.0i   3.0 + 0.0i
 P2:
  4.0 + 0.0i   5.0 + 0.0i   6.0 + 0.0i
 P3:
  4.0 + 0.0i  13.0 + 0.0i  28.0 + 0.0i  27.0 + 0.0i  18.0 + 0.0i
```

If any vector overlaps a writable vector, either because of argument aliasing or ill-chosen values of the various `INC` arguments, the results are undefined and can vary from one run to the next.

The most common form of the computation, and the case that executes fastest, is applying a filter vector `X` to a series of vectors stored in the columns of `Y` with the result placed into the columns of `Z`. In that case, `INCX` = 1, `INC1Y` = 1, `INC2Y` ≥ `NY`, `INC1Z` = 1, `INC2Z` ≥ `NZ`. Another common form is applying a filter vector `X` to a series of vectors stored in the rows of `Y` and store the result in the row of `Z`, in which case `INCX` = 1, `INC1Y` ≥ `NY`, `INC2Y` = 1, `INC1Z` ≥ `NZ`, and `INC2Z` = 1.

Convolution can be used to compute the products of polynomials. CODE EXAMPLE 5-11 uses SCNVCOR to compute the product of $1 + 2x + 3x^2$ and $4 + 5x + 6x^2$.

CODE EXAMPLE 5-11 One-Dimensional Convolution Using Fourier Transform Method and REAL Data

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

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

CODE EXAMPLE 5-12 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 5-12 Convolution Used to Compute the Product of a Vector and Circulant Matrix

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

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

CODE EXAMPLE 5-13 Two-Dimensional Convolution Using Direct Method

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

References

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

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

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

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

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

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

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

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

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

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

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

CHAPTER **6**

Interval BLAS Routines

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 `SFFTC` routine, type **`man -s 3P sfftc`**. Routine names must be lowercase.

For more information on the non-interval version of the BLAS library, see the document *Basic Linear Algebra Subprogram Technical (BLAST) Forum Standard*, available at `http://www.netlib.org/blas/blast-forum/`.

Note – For the Forte Developer 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.

Intervals

Intervals have a dual identity as intervals of *numbers* and as *sets* of numbers. The empty interval ∅ 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 `[empty]`. 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.

IBLAS Routine Names

This section summarizes IBLAS naming conventions derived from the BLAS specification. "Language Bindings" on page 108 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 6-2 through TABLE 6-4.
- Matrix-Vector Operations Table, listed in TABLE 6-5.
- Matrix Operations Tables, listed in TABLE 6-6 through TABLE 6-8.

New interval-specific routines are grouped into:

- Set Operations on Vectors, listed in TABLE 6-9.
- Set Operations on Matrices, listed in TABLE 6-10.
- Utility Functions of Vectors, listed in TABLE 6-11.
- Utility Functions of Matrices, listed in TABLE 6-12.

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 6-1 lists the IBLAS prefixes and matrix types.

TABLE 6-1 IBLAS Prefixes and Matrix Types

Prefix	Matrix Type
`GE`	General
`GB`	General Banded
`SY`	Symmetric
`SB`	Symmetric Banded
`SP`	Symmetric Packed
`TR`	Triangular
`TB`	Triangular Banded
`TP`	Triangular Packed

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 6-9 through TABLE 6-12.

Fortran Interface

The IBLAS Fortran bindings are implemented in a module. Its interface block defines the default interval data type to be `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 *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 Forte Developer `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 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.

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 (AXPBY_I) routine.

```
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:

```
<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.

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 6-3 and TABLE 6-9 through TABLE 6-12. For clarity, lowercase and uppercase Fortran variable names are used to distinguish point from interval types. See TABLE A-11 for an alphabetical list of all the IBLAS routines.

TABLE 6-2 Vector Reductions

Name	Function
DOT_I	Dot Product
NORM_I	Vector Norms
SUM_I	Sum
AMIN_VAL_I	Minimum Absolute Value and Location
AMAX_VAL_I	Maximum Absolute Value and Location
SUMSQ_I	Scaled Sum of Squares and Update

TABLE 6-3 Add or Cancel Vectors

Name	Operation	f95 Equivalent
RSCALE_I	Reciprocally Scale Vector	
AXPBY_I	Add Scaled Vectors and Update	
WAXPBY_I	Add Scaled Vectors	
CANCEL_I	Cancel Scaled Vectors and Update	`Y = a*X .DSUB. b*Y`
WCANCEL_I	Cancel Scaled Vectors	`W = a*X .DSUB. b*Y`
SUMSQ_I	Scaled Sum of Squares and Update	

TABLE 6-4 Vector Movements

Name	Operation
COPY_I	Vector Copy
SWAP_I	Vector Swap
PERMUTE_I	Permute Vector and Update

TABLE 6-5 Matrix-Vector Operations

Name	Operation
{GE,GB}MV_I	General Matrix-Vector Product and Update
{SY,SB,SP}MV_I	Symmetric Matrix-Vector Product and Update
{TR,TB,TP}MV_I	Triangular Matrix-Vector Product and Update
{TR,TB,TP}SV_I	Triangular Matrix Solve and Update
GER_I	General-Matrix Rank-One Update
{SY,SP}R_I	Symmetric-Matrix Rank-One Update

TABLE 6-6 O(n^2) Matrix Operations

Name	Operation
{GE, GB, SY, SB, SP, TR, TB, TP}_NORM_I	Matrix Norms
{GE, GB}_DIAG_SCALE_I	Scale General Matrix Rows or Columns and Update
{GE, GB}_LRSCALE_I	Scale General Matrix Rows and Columns and Update
{SY, SB, SP}_LRSCALE_I	Scale Symmetric Matrix Rows and Columns and Update
{GE, GB, SY, SB, SP, TR, TB, TP}_ACC_I	Add Scaled Matrices and Update
{GE, GB, SY, SB, SP, TR, TB, TP}_ADD_I	Add Scaled Matrices

TABLE 6-7 O(n^3) Matrix Operations

Name	Operation
GEMM_I	General Matrix-Matrix Product and Update
SYMM_I	Symmetric-General Matrix-Matrix Product and Update
TRMM_I	Triangular-General Matrix-Matrix Product and Update
TRSM_I	Triangular Matrix Solve

TABLE 6-8 Matrix Movements

Name	Operation
{GE,GB,SY,SB,SP,TR,TB,TP}_COPY_I	Copy Matrix
GE_TRANS_I	Transpose Matrix
GE_PERMUTE_I	Permute Matrix

In some cases the IBLAS routine names listed in TABLE 6-9 through TABLE 6-11 differ from the names of the IBLAS routines included in the Sun Performance Library Forte Developer 7 release. When the names differ, the name of the IBLAS routine included with the Forte Developer 7 release is listed in parenthesis. See the Interval BLAS white paper for more information on the IBLAS routine names.

TABLE 6-9 Vector Set Operations

Name	Operation	f95 Equivalent
ENCLOSEV_I (ENCV_I)	Enclose Vector Test	X.SB.Y
INTERIORV_I	Vector Interior Test	X.INT.Y
DISJOINTV_I (DISJV_I)	Disjoint Vector Test	X.DJ.Y
INTERSECTV_I (INTERV_I)	Intersect Vectors and Update	Y = X.IX.Y
WINTERSECTV_I (WINTERV_I)	Intersect Vectors	W = X.IX.Y
HULLV_I	Hull of Vectors and Update	Y = X.IH.Y
WHULLV_I	Hull of Vectors	W = X.IH.Y

TABLE 6-10 Matrix Set Operations

Prefix	Name	Operation	f95 Equivalent
	ENCLOSEM_I (ENCM_I)	Enclose Matrix Test	A.SB.B
	INTERIORM_I	Matrix Interior Test	A.INT.B
	DISJOINTM_I (DISJM_I)	Disjoint Matrix Test	A.DJ.B
{GE, GB, SY, SB, SP, TR, TB, TP}_	INTERSECTM_I (INTERM_I)	Intersect Matrices and Update	B = X.IX.B
	WINTERSECTM_I (WINTERM_I)	Intersect Matrices	W = X.IX.B
	HULLM_I	Hull of Matrices and Update	B = X.IH.B
	WHULLM_I	Hull of Matrices	W = X.IH.B

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

TABLE 6-11 Vector Utilities

Name	Operation	f95 Equivalent
EMPTYV_I (EMPTYELEV_I)	Empty Vector Element Test and Location (Empty Vector Element Location, *only*)	ISEMPTY(X)
INFV_I	Vector Infimum	v = INF(X)
SUPV_I	Vector Supremum	v = SUP(X)
MIDV_I	Vector Midpoint	v = MID(X)
WIDTHV_I	Vector Width	v = WID(X)
INTERVALV_I (CONSTRUCTV_I)	Vector Type Conversion to Interval	X = INTERVAL(u,v)

TABLE 6-12 Matrix Utilities

Prefix	Name	Operation	f95 Equivalent
	EMPTYM_I	Empty Matrix Element Test and Location	ISEMPTY(A)
	(EMPTYELEM_I)	(Empty Vector Element Location, *only*)	
	INFM_I	Matrix Infimum	c = INF(A)
{GE, GB, SY, SB, SP, TR, TB, TP}_	SUPM_I	Matrix Supremum	c = SUP(A)
	MIDM_I	Matrix Midpoint	c = MID(A)
	WIDTHM_I	Matrix Width	c = WID(A)
	INTERVALM_I (CONSTRUCTM_I)	Matrix Type Conversion to Interval	A = INTERVAL(b,c)

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

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.

APPENDIX **A**

Sun Performance Library Routines

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

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

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

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

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

LAPACK Routines

TABLE A-1 lists the Sun Performance Library LAPACK routines.

TABLE A-1 LAPACK (Linear Algebra Package) Routines

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

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

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

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

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

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

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

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

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

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

Routine	Function
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 or ZHPTRS	Solves a complex Hermitian indefinite matrix in packed storage, using the factorization computed by CHPTRF or ZHPTRF.
Upper Hessenberg Matrix	
*x*HSEIN	Computes right and/or left eigenvectors of upper Hessenberg matrix using inverse iteration.
*x*HSEQR	Computes eigenvectors and Shur factorization of upper Hessenberg matrix using multishift QR algorithm.
Upper Hessenberg Matrix-Generalized Problem (Hessenberg and Triangular Matrix)	
*x*HGEQZ	Implements single-/double-shift version of QZ method for finding the generalized eigenvalues of the equation det(A - w(i) * B) = 0.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Routine	Function
CUNMBR or ZUNMBR	Multiplies a general matrix with the unitary transformation matrix reduced to bidiagonal form, as determined by CGEBRD or ZGEBRD.
CUNMHR or ZUNMHR	Multiplies a general matrix by the unitary matrix reduced to Hessenberg form by CGEHRD or ZGEHRD.
CUNMLQ or ZUNMLQ	Multiplies a general matrix by the unitary matrix from an LQ factorization, as returned by CGELQF or ZGELQF.
CUNMQL or ZUNMQL	Multiplies a general matrix by the unitary matrix from a QL factorization, as returned by CGEQLF or ZGEQLF.
CUNMQR or ZUNMQR	Multiplies a general matrix by the unitary matrix from a QR factorization, as returned by CGEQRF or ZGEQRF.
CUNMRQ or ZUNMRQ	Multiplies a general matrix by the unitary matrix from an RQ factorization, as returned by CGERQF or ZGERQF.
CUNMRZ or ZUNMRZ	Multiplies a general matrix by the unitary matrix from an RZ factorization, as returned by CTZRZF or ZTZRZF.
CUNMTR or ZUNMTR	Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHETRD or ZHETRD.
Unitary Matrix in Packed Storage	
CUPGTR or ZUPGTR	Generates the unitary transformation matrix from a tridiagonal matrix determined by CHPTRD or ZHPTRD.
CUPMTR or ZUPMTR	Multiplies a general matrix by the unitary transformation matrix reduced to tridiagonal form by CHPTRD or ZHPTRD.

BLAS1 Routines

TABLE A-2 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

Routine	**Function**
SASUM, DASUM, SCASUM, DZASUM	Sum of the absolute values of a vector
*x*AXPY	Product of a scalar and vector plus a vector
*x*COPY	Copy a vector
SDOT, DDOT, DSDOT, SDSDOT, CDOTU, ZDOTU, DQDOTA, DQDOTI	Dot product (inner product)
CDOTC, ZDOTC	Dot product conjugating first vector
SNRM2, DNRM2, SCNRM2, DZNRM2	Euclidean norm of a vector
*x*ROTG	Set up Givens plane rotation
*x*ROT, CSROT, ZDROT	Apply Given's plane rotation
SROTMG, DROTMG	Set up modified Given's plane rotation
SROTM, DROTM	Apply modified Given's rotation
ISAMAX, DAMAX, ICAMAX, IZAMAX	Index of element with maximum absolute value
*x*SCAL, CSSCAL, ZDSCAL	Scale a vector
*x*SWAP	Swap two vectors
CVMUL, ZVMUL	Compute scaled product of complex vectors

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

Routine	Function
*x*GBMV	Product of a matrix in banded storage and a vector
*x*GEMV (P)	Product of a general matrix and a vector
SGER (P), DGER (P), CGERC (P), ZGERC (P), CGERU (P), ZGERU (P)	Rank-1 update to a general matrix
CHBMV, ZHBMV	Product of a Hermitian matrix in banded storage and a vector
CHEMV (P), ZHEMV (P)	Product of a Hermitian matrix and a vector
CHER (P), ZHER (P)	Rank-1 update to a Hermitian matrix
CHER2, ZHER2	Rank-2 update to a Hermitian matrix
CHPMV (P), ZHPMV (P)	Product of a Hermitian matrix in packed storage and a vector
CHPR, ZHPR	Rank-1 update to a Hermitian matrix in packed storage
CHPR2, ZHPR2	Rank-2 update to a Hermitian matrix in packed storage
SSBMV, DSBMV	Product of a symmetric matrix in banded storage and a vector
SSPMV (P), DSPMV (P)	Product of a Symmetric matrix in packed storage and a vector
SSPR, DSPR	Rank-1 update to a real symmetric matrix in packed storage
SSPR2 (P), DSPR2 (P)	Rank-2 update to a real symmetric matrix in packed storage
SSYMV, DSYMV	Product of a symmetric matrix and a vector
SSYR (P), DSYR (P)	Rank-1 update to a real symmetric matrix
SSYR2 (P), DSYR2 (P)	Rank-2 update to a real symmetric matrix
*x*TBMV	Product of a triangular matrix in banded storage and a vector
*x*TBSV	Solution to a triangular system in banded storage of linear equations
*x*TPMV	Product of a triangular matrix in packed storage and a vector
*x*TPSV	Solution to a triangular system of linear equations in packed storage
*x*TRMV (P)	Product of a triangular matrix and a vector
*x*TRSV	Solution to a triangular system of linear equations

BLAS3 Routines

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

TABLE A-4 BLAS3 (Basic Linear Algebra Subprograms, Level 3) Routines

Routine	Function
x`GEMM` (P)	Product of two general matrices
`CHEMM` (P) or `ZHEMM` (P)	Product of a Hermitian matrix and a general matrix
`CHERK` (P) or `ZHERK` (P)	Rank-k update of a Hermitian matrix
`CHER2K` (P) or `ZHER2K` (P)	Rank-2k update of a Hermitian matrix
x`SYMM` (P)	Product of a symmetric matrix and a general matrix
x`SYRK` (P)	Rank-k update of a symmetric matrix
x`SYR2K` (P)	Rank-2k update of a symmetric matrix
x`TRMM` (P)	Product of a triangular matrix and a general matrix
x`TRSM` (P)	Solution for a triangular system of equations

Sparse BLAS Routines

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

TABLE A-5 Sparse BLAS Routines

Routines	Function
x`AXPYI`	Adds a scalar multiple of a sparse vector *X* to a full vector *Y*.
x`BCOMM` (P)	Block coordinate matrix-matrix multiply.
x`BDIMM` (P)	Block diagonal format matrix-matrix multiply.
x`BDISM` (P)	Block Diagonal format triangular solve.
x`BELMM` (P)	Block Ellpack format matrix-matrix multiply.
x`BELSM` (P)	Block Ellpack format triangular solve.
x`BSCMM` (P)	Block compressed sparse column format matrix-matrix multiply.

TABLE A-5 Sparse BLAS Routines *(Continued)*

Routines	Function
*x*BSCSM (P)	Block compressed sparse column format triangular solve.
*x*BSRMM (P)	Block compressed sparse row format matrix-matrix multiply.
*x*BSRSM (P)	Block compressed sparse row format triangular solve.
*x*COOMM (P)	Coordinate format matrix-matrix multiply.
*x*CSCMM (P)	Compressed sparse column format matrix-matrix multiply
*x*CSCSM (P)	Compressed sparse column format triangular solve
*x*CSRMM (P)	Compressed sparse row format matrix-matrix multiply.
*x*CSRSM (P)	Compressed sparse row format triangular solve.
*x*DIAMM (P)	Diagonal format matrix-matrix multiply.
*x*DIASM (P)	Diagonal format triangular solve.
SDOTI, DDOTI, CDOTUI, or ZDOTUI	Computes the dot product of a sparse vector and a full vector.
CDOTCI, or ZDOTCI	Computes the conjugate dot product of a sparse vector and a full vector.
*x*ELLMM (P)	Ellpack format matrix-matrix multiply.
*x*ELLSM (P)	Ellpack format triangular solve.
*x*CGTHR	Given a full vector, creates a sparse vector and corresponding index vector.
*x*CGTHRZ	Given a full vector, creates a sparse vector and corresponding index vector and zeros the full vector.
*x*JADMM (P)	Jagged diagonal matrix-matrix multiply.
SJADRP or DJADRP	Right permutation of a jagged diagonal matrix.
*x*JADSM (P)	Jagged diagonal triangular solve.
SROTI or DROTI	Applies a Givens rotation to a sparse vector and a full vector.
*x*CSCTR	Given a sparse vector and corresponding index vector, puts those elements into a full vector.
*x*SKYMM (P)	Skyline format matrix-matrix multiply.
*x*SKYSM (P)	Skyline format triangular solve.
*x*VBRMM (P)	Variable block sparse row format matrix-matrix multiply.
*x*VBRSM (P)	Variable block sparse row format triangular solve.

Sparse Solver Routines

TABLE A-6 lists the Sun Performance Library sparse solver routines.

TABLE A-6 Sparse Solver Routines

Routines	Function
SGSSFS, DGSSFS, CGSSFS, or ZGSSFS	One call interface to sparse solver.
SGSSIN, DGSSIN, CGSSIN, or ZGSSIN	Sparse solver initialization.
SGSSOR, DGSSOR, CGSSOR, or ZGSSOR	Fill reducing ordering and symbolic factorization.
SGSSFA, DGSSFA, CGSSFA, or ZGSSFA	Matrix value input and numeric factorization.
SGSSSL, DGSSSL, CGSSSL, or ZGSSSL	Triangular solve.
SGSSUO, DGSSUO, CGSSUO, or ZGSSUO	Sets user-specified ordering permutation.
SGSSRP, DGSSRP, CGSSRP, or ZGSSRP	Returns permutation used by solver.
SGSSCO, DGSSCO, CGSSCO, or ZGSSCO	Returns condition number estimate of coefficient matrix.
SGSSDA, DGSSDA, CGSSDA, or ZGSSDA	De-allocates sparse solver.
SGSSPS, DGSSPS, CGSSPS, or ZGSSPS	Prints solver statistics.

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 legacy FFT routines and man pages for the routines are still included to maintain compatibility with existing codes, but the routines are no longer supported. For information on using the legacy FFT routines, see the section 3p man pages.

TABLE A-7 shows the mapping between the Forte Developer FFT routines and the corresponding FFTPACK and VFFTPACK routines. (P) denotes routines that are parallelized.

TABLE A-7 FFT Routines

Routine	Replaces	Function
`CFFTC` (P)	`CFFTI` `CFFTF` (P) `CFFTB` (P)	Initialize the trigonometric weight and factor tables or compute the one-dimensional forward or inverse FFT of a complex sequence.
`CFFTC2` (P)	`CFFT2I` `CFFT2F` (P) `CFFT2B` (P)	Initialize the trigonometric weight and factor tables or compute the two-dimensional forward or inverse FFT of a two-dimensional complex array.
`CFFTC3` (P)	`CFFT3I` `CFFT3F` (P) `CFFT3B` (P)	Initialize the trigonometric weight and factor tables or compute the three-dimensional forward or inverse FFT of three-dimensional complex array.
`CFFTCM` (P)	`VCFFTI` `VCFFTF` (P) `VCFFTB` (P)	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.

`CFFTS`	`RFFTI`, `RFFTB`
	`EZFFTI`, `EZFFTB`
		Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a complex sequence.

`CFFTS2`	`RFFT2I` `RFFT2B`	Initialize the trigonometric weight and factor tables or compute the two-dimensional inverse FFT of a two-dimensional complex array.
`CFFTS3` (P)	`RFFT3I` `RFFT3B`	Initialize the trigonometric weight and factor tables or compute the three-dimensional inverse FFT of three-dimensional complex array.

TABLE A-7 FFT Routines *(Continued)*

Routine	Replaces	Function
`CFFTSM`	`VRFFTI` `VRFFTB` (P)	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.

`DFFTZ`	`DFFTI`, `DFFTF`
	`DEZFFTI`, `DEZFFTF`
		Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a double precision sequence.

`DFFTZ2`	`DFFT2I` `DFFT2F`	Initialize the trigonometric weight and factor tables or compute the two-dimensional forward FFT of a two-dimensional double precision array.
`DFFTZ3` (P)	`DFFT3I` `DFFT3F`	Initialize the trigonometric weight and factor tables or compute the three-dimensional forward FFT of three-dimensional double precision array.
`DFFTZM`	`VDFFTI` `VDFFTF` (P)	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.

`SFFTC`	`RFFTI`, `RFFTF`
	`EZFFTI`, `EZFFTF`
		Initialize the trigonometric weight and factor tables or compute the one-dimensional forward FFT of a real sequence.

`SFFTC2`	`RFFT2I` `RFFT2F`	Initialize the trigonometric weight and factor tables or compute the two-dimensional forward FFT of a two-dimensional real array.
`SFFTC3` (P)	`RFFT3I` `RFFT3F`	Initialize the trigonometric weight and factor tables or compute the three-dimensional forward FFT of three-dimensional real array.
`SFFTCM`	`VRFFTI` `VRFFTF` (P)	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.

`ZFFTD`	`DFFTI`, `DFFTB`
	`DEZFFTI`, `DEZFFTB`
		Initialize the trigonometric weight and factor tables or compute the one-dimensional inverse FFT of a double complex sequence.

`ZFFTD2`	`DFFT2I` `DFFT2B`	Initialize the trigonometric weight and factor tables or compute the two-dimensional inverse FFT of a two-dimensional double complex array.
`ZFFTD3` (P)	`DFFT3I` `DFFT3B`	Initialize the trigonometric weight and factor tables or compute the three-dimensional inverse FFT of three-dimensional double complex array.
`ZFFTDM`	`VDFFTI` `VDFFTB` (P)	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.

TABLE A-7 FFT Routines *(Continued)*

Routine	Replaces	Function
`ZFFTZ` (P)	`ZFFTI` `ZFFTF` (P) `ZFFTB` (P)	Initialize the trigonometric weight and factor tables or compute the one-dimensional forward or inverse FFT of a double complex sequence.
`ZFFTZ2` (P)	`ZFFT2I` `ZFFT2F` (P) `ZFFT2B` (P)	Initialize the trigonometric weight and factor tables or compute the two-dimensional forward or inverse FFT of a two-dimensional double complex array.
`ZFFTZ3` (P)	`ZFFT3I` `ZFFT3F` (P) `ZFFT3B` (P)	Initialize the trigonometric weight and factor tables or compute the three-dimensional forward or inverse FFT of three-dimensional double complex array.
`ZFFTZM` (P)	`VZFFTI` `VZFFTF` (P) `VZFFTB` (P)	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 double complex array.

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-8 lists the Sun Performance Library sine and cosine transform routines.

TABLE A-8 Sine and Cosine Transform Routines

Routine	Function
`COSQB`, `DCOSQB`, `VCOSQB`, `VDCOSQB`	Cosine quarter-wave synthesis.
`COSQF`, `DCOSQF`, `VCOSQF`, `VDCOSQF`	Cosine quarter-wave transform.
`COSQI`, `DCOSQI`, `VCOSQI`, `VDCOSQI`	Initialize cosine quarter-wave transform and synthesis.
`COST`, `DCOST`, `VCOST, VDCOST`	Cosine even-wave transform.
`COSTI`, `DCOSTI`, `VCOSTI`, `VDCOSTI`	Initialize cosine even-wave transform.
`SINQB`, `DSINQB`, `VSINQB`, `VDSINQB`	Sine quarter-wave synthesis.

TABLE A-8 Sine and Cosine Transform Routines *(Continued)*

Routine	Function
SINQF, DSINQF, VSINQF, VDSINQF	Sine quarter-wave transform.
SINQI, DSINQI, VSINQI, VDSINQI	Initialize sine quarter-wave transform and synthesis.
SINT, DSINT, VSINT, VDSINT	Sine odd-wave transform.
SINTI, DSINT, VSINTI, VDSINTI	Initialize sine odd-wave transform.

Convolution and Correlation Routines

TABLE A-9 lists the Sun Performance Library convolution and correlation routines.

TABLE A-9 Convolution and Correlation Routines

Routines	Function
xCNVCOR	Computes convolution or correlation
xCNVCOR2	Computes two-dimensional convolution or correlation

Miscellaneous Signal Processing Routines

TABLE A-10 lists the miscellaneous Sun Performance Library signal processing routines.

TABLE A-10 Convolution and Correlation Routines

Routines	Function
RFFTOPT, DFFTOPT, CFFTOPT, ZFFTOPT	Compute the length of the closest FFT
SWIENER or DWEINER	Performs Wiener deconvolution of two signals
xTRANS	Transposes array

Interval BLAS (IBLAS) Routines

Sun Performance Library includes the interval BLAS routines listed in TABLE A-11, which operate on interval scalars, interval vectors, and interval matrices (dense, banded, and triangular).

TABLE A-11 Interval BLAS Routines

Routine	Function
`amax_val_i`	Max absolute value and location.
`amin_val_i`	Min absolute value and location.
`axpby_i`	Scaled vector accumulation.
`cancel_i`	Scaled cancellation.
`constructv_i`	Constructs an interval vector.
`copy_i`	Interval vector copy.
`disjv_i`	Checks if two interval vectors disjoint.
`dot_i`	Scaled dot product of two interval vectors.
`emptyelev_i`	Empty entry and its location.
`encv_i`	Check if an interval vector is enclosed in another.
`fpinfo_i`	Environmental enquiry.
`gbmv_i`	Interval matrix-vector multiplication.
`gb_acc_i`	General band matrix accumulation and scale.
`gb_add_i`	General band matrix add and scale.
`gb_constructm_i`	Constructs an interval matrix from two floating point matrices.
`gb_copy_i`	General band interval matrix copy.
`gb_diag_scale_i`	Diagonal scaling of an interval matrix.
`gb_disjm_i`	If two interval matrices are disjoint.
`gb_emptyelem_i`	Empty entry and its location.
`gb_encm_i`	If an interval matrix is enclosed in another.
`gb_hullm_i`	Convex hull of two interval matrices.
`gb_infm_i`	Left endpoint of an interval matrix.
`gb_interiorm_i`	If an interval matrix is in interior of another.
`gb_interm_i`	Intersection of two interval matrices.
`gb_lrscale_i`	Two-sided diagonal scaling.

TABLE A-11 Interval BLAS Routines *(Continued)*

Routine	Function
`gb_midm_i`	Midpoint matrix of an interval matrix.
`gb_norm_i`	General band interval matrix norms.
`gb_supm_i`	Right endpoint of an interval matrix.
`gb_whullm_i`	Convex hull of two interval matrices.
`gb_widthm_i`	Elementwise width of an interval matrix.
`gb_winterm_i`	Intersection of two interval matrices.
`gemm_i`	General interval matrix product.
`gemv_i`	General interval matrix and vector multiplication.
`ger_i`	Rank one update.
`ge_acc_i`	General matrix accumulation and scale.
`ge_add_i`	General interval matrix add and scale.
`ge_constructm_i`	Constructs an interval matrix from two floating point matrices.
`ge_copy_i`	General interval matrix copy.
`ge_diag_scale_i`	Diagonal scaling an interval matrix.
`ge_disjm_i`	If two interval matrices are disjoint.
`ge_emptyelem_i`	Empty entry and its location.
`ge_encm_i`	If an interval matrix is enclosed in another.
`ge_hullm_i`	Convex hull of two interval matrices.
`ge_infm_i`	Left endpoint of an interval matrix.
`ge_interiorm_i`	If an interval matrix is in interior of another.
`ge_interm_i`	Intersection of two interval matrices.
`ge_lrscale_i`	Two-sided diagonal scaling.
`ge_midm_i`	Midpoint matrix of an interval matrix.
`ge_norm_i`	General interval matrix norms.
`ge_permute_i`	Permute an general interval matrix.
`ge_supm_i`	Right endpoint of an interval matrix.
`ge_trans_i`	Matrix transposition.
`ge_whullm_i`	Convex hull of two interval matrices.
`ge_widthm_i`	Elementwise width of an interval matrix.
`ge_winterm_i`	Intersection of two interval matrices.

TABLE A-11 Interval BLAS Routines *(Continued)*

Routine	Function
`hullv_i`	Convex hull of an interval vector with another.
`infv_i`	The left endpoint of an interval vector.
`interiorv_i`	If an interval vector is in the interior of another.
`interv_i`	Intersection of an interval vector with another.
`midv_i`	The approximate midpoint of an interval vector.
`norm_i`	Interval vector norms.
`permute_i`	Permute interval vector.
`rscale_i`	Reciprocal scale of an interval vector.
`sbmv_i`	Interval symmetric matrix vector product.
`sb_acc_i`	Symmetric band matrix accumulation and scale.
`sb_add_i`	Symmetric band matrix add and scale.
`sb_constructm_i`	Constructs an interval matrix from two floating point matrices.
`sb_copy_i`	Symmetric band interval matrix copy.
`sb_disjm_i`	If two interval matrices are disjoint.
`sb_emptyelem_i`	Empty entry and its location.
`sb_encm_i`	If an interval matrix is enclosed in another.
`sb_hullm_i`	Convex hull of two interval matrices.
`sb_infm_i`	Left endpoint of an interval matrix.
`sb_interiorm_i`	If an interval matrix is in interior of another.
`sb_interm_i`	Intersection of two interval matrices.
`sb_lrscale_i`	Two-sided diagonal scaling.
`sb_midm_i`	Midpoint matrix of an interval matrix.
`sb_norm_i`	Symmetric band interval matrix norms.
`sb_supm_i`	Right endpoint of an interval matrix.
`sb_whullm_i`	Convex hull of two interval matrices.
`sb_widthm_i`	Elementwise width of an interval matrix.
`sb_winterm_i`	Intersection of two interval matrices.
`spmv_i`	Interval symmetric matrix vector product.
`spr_i`	Symmetric rank one update.
`sp_acc_i`	Symmetric packed matrix accumulation and scale.

TABLE A-11 Interval BLAS Routines *(Continued)*

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

TABLE A-11 Interval BLAS Routines *(Continued)*

Routine	Function
sy_encm_i	If an interval matrix is enclosed in another.
sy_hullm_i	Convex hull of two interval matrices.
sy_infm_i	Left endpoint of an interval matrix.
sy_interiorm_i	If an interval matrix is in interior of another.
sy_interm_i	Intersection of two interval matrices.
sy_lrscale_i	Two-sided diagonal scaling.
sy_midm_i	Midpoint matrix of an interval matrix.
sy_norm_i	Symmetric interval matrix norms.
sy_supm_i	Right endpoint of an interval matrix.
sy_whullm_i	Convex hull of two interval matrices.
sy_widthm_i	Elementwise width of an interval matrix.
sy_winterm_i	Intersection of two interval matrices.
tbmv_i	Interval triangular matrix vector product.
tbsv_i	Interval triangular solve with a vector.
tb_acc_i	Matrix accumulation and scale.
tb_add_i	Triangular band matrix add and scale.
tb_constructm_i	Constructs an interval matrix from two floating point matrices.
tb_copy_i	Triangular band interval matrix copy.
tb_disjm_i	If two interval matrices are disjoint.
tb_emptyelem_i	Empty entry and its location.
tb_encm_i	If an interval matrix is enclosed in another.
tb_hullm_i	Convex hull of two interval matrices.
tb_infm_i	Left endpoint of an interval matrix.
tb_interiorm_i	If an interval matrix is in interior of another.
tb_interm_i	Intersection of two interval matrices.
tb_midm_i	Midpoint matrix of an interval matrix.
tb_norm_i	Triangular band interval matrix norms.
tb_supm_i	Right endpoint of an interval matrix.
tb_whullm_i	Convex hull of two interval matrices.
tb_widthm_i	Elementwise width of an interval matrix.

TABLE A-11 Interval BLAS Routines *(Continued)*

Routine	Function
`tb_winterm_i`	Intersection of two interval matrices.
`tpmv_i`	Interval triangular matrix vector product.
`tpsv_i`	Interval triangular solve with a vector.
`tp_acc_i`	Matrix accumulation and scale.
`tp_add_i`	Triangular packed matrix add and scale.
`tp_constructm_i`	Constructs an interval matrix from two floating point matrices.
`tp_copy_i`	Triangular packed interval matrix copy.
`tp_disjm_i`	If two interval matrices are disjoint.
`tp_emptyelem_i`	Empty entry and its location.
`tp_encm_i`	If an interval matrix is enclosed in another.
`tp_hullm_i`	Convex hull of two interval matrices.
`tp_infm_i`	Left endpoint of an interval matrix.
`tp_interiorm_i`	If an interval matrix is in interior of another.
`tp_interm_i`	Intersection of two interval matrices.
`tp_midm_i`	Midpoint matrix of an interval matrix.
`tp_norm_i`	Triangular packed interval matrix norms.
`tp_supm_i`	Right endpoint of an interval matrix.
`tp_whullm_i`	Convex hull of two interval matrices.
`tp_widthm_i`	Elementwise width of an interval matrix.
`tp_winterm_i`	Intersection of two interval matrices.
`trmm_i`	Triangular interval matrix matrix product.
`trmv_i`	Interval triangular matrix vector product.
`trsm_i`	Interval triangular solve.
`trsv_i`	Interval triangular solve with a vector.
`tr_acc_i`	Matrix accumulation and scale.
`tr_add_i`	Triangular matrix add and scale.
`tr_constructm_i`	Constructs an interval matrix from two floating point matrices.
`tr_copy_i`	Triangular interval matrix copy.
`tr_disjm_i`	If two interval matrices are disjoint.
`tr_emptyelem_i`	Empty entry and its location.

TABLE A-11 Interval BLAS Routines *(Continued)*

Routine	Function
`tr_encm_i`	If an interval matrix is enclosed in another.
`tr_hullm_i`	Convex hull of two interval matrices.
`tr_infm_i`	Left endpoint of an interval matrix.
`tr_interiorm_i`	If an interval matrix is in interior of another.
`tr_interm_i`	Intersection of two interval matrices.
`tr_midm_i`	Midpoint matrix of an interval matrix.
`tr_norm_i`	Triangular interval matrix norms.
`tr_supm_i`	Right endpoint of an interval matrix.
`tr_whullm_i`	Convex hull of two interval matrices.
`tr_widthm_i`	Elementwise width of an interval matrix.
`tr_winterm_i`	Intersection of two interval matrices.
`waxpby_i`	Scaled vector addition.
`wcancel_i`	Scaled cancellation.
`whullv_i`	Convex hull of an interval vector with another.
`widthv_i`	The elementwise width of an interval vector.
`winterv_i`	Intersection of an interval vector with another.

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

Sort Routines

TABLE A-12 lists the Sun Performance Library sort routines.

TABLE A-12 Sort Routines

Routines	**Function**
`BLAS_DSORT`	Sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm.
`BLAS_DSORTV`	Sorts a real (double precision) vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.
`BLAS_DPERMUTE`	Permutes a real (double precision) array in terms of the permutation vector P, output by `DSORTV`.
`BLAS_ISORT`	Sorts an integer vector X in increasing or decreasing order using quick sort algorithm.
`BLAS_ISORTV`	Sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.
`BLAS_IPERMUTE`	Permutes an integer array in terms of the permutation vector P, output by `DSORTV.`
`BLAS_SSORT`	Sorts a real vector X in increasing or decreasing order using quick sort algorithm.
`BLAS_SSORTV`	Sorts a real vector X in increasing or decreasing order using quick sort algorithm and overwrite P with the permutation vector.
`BLAS_SPERMUTE`	Permutes a real array in terms of the permutation vector P, output by `DSORTV`.

Index