HP MLIB User's Guide

VECLIB and LAPACK

First Edition

HP Part No. B6061-96010
HP MLIB
December 1999

Printed in USA

Revision History

Edition: First

Document Number: B6061-96010
Remarks: Released December 1999 with HP MLIB software version B.07.00. This *HP MLIB User's Guide* contains both HP VECLIB and HP LAPACK information. This document supercedes both the *HP MLIB VECLIB User's Guide*, fifth edition (B6061-96006) and the *HP MLIB LAPACK User's Guide*, sixth edition (B6061-96005).

Notice

Contents

Part 2 HP LAPACK

Table of contents

Figures

List of Figures

Tables

List of Tables

Preface

Hewlett-Packard's high-performance math libraries (HP MLIB) help you speed development of applications and shorten execution time of long-running technical applications.

HP MLIB is a collection of subprograms optimized for use on HP servers and workstations, providing mathematical software and computational kernels for engineering and scientific applications. HP MLIB can be used on HP-UX systems ranging from single-processor workstations to multiprocessor high-end servers such as the V- and N-Class. HP MLIB is optimized for HP PA-RISC 2.0 processors. HP MLIB has two components; LAPACK and VECLIB.

VECLIB

HP VECLIB contains robust callable subprograms. Together with a subset of the BLAS Standard subroutines, HP MLIB supports the legacy BLAS, a collection of routines for the solution of sparse symmetric systems of equations, a collection of commonly used Fast Fourier Transforms (FFTs), and convolutions. Although VECLIB was designed for use with Fortran programs, C programs can call VECLIB subprograms, as described in Appendix A. Refer to Part 1 of this manual for HP VECLIB information.

Throughout this document there are references to legacy BLAS and BLAS Standard routines. Legacy BLAS routines include Basic Linear Algebra Subprograms (BLAS), that is the level 1, 2, and 3 BLAS, as well as the Sparse BLAS.

A BLAS standardization effort began with a BLAS Technical (BLAST) Forum meeting in November 1995 at the University of Tennessee. The efforts of the BLAST Forum resulted in a BLAS Standard specification in 1999. BLAS Standard routines refer to routines as defined by this BLAS Standard specification. HP MLIB 7.0 supports a subset of BLAS Standard routines. Refer to Chapter 2, "Basic Vector Operations," and Chapter 3, "Basic Matrix Operations," for details about supported subprograms.

LAPACK

HP Linear Algebra Package (LAPACK) is a collection of subprograms that provide mathematical software for applications involving linear equations, least squares, eigenvalue problems, and the singular value decomposition. LAPACK is designed to supersede the linear equation and eigenvalue packages, LINPACK and EISPACK. The National Science Foundation, the Defense Advanced Research Projects Agency, and the Department of Energy supported the development of the public-domain version of LAPACK, from which the HP version was derived.

HP LAPACK fully conforms with public domain version 3.0 of LAPACK in all user-visible usage conventions. Refer to Part 2 of this manual for information specific to HP LAPACK. To supplement the HP specific information provided in Part 2 of this document, refer to the standard *LAPACK Users' Guide*. You can access the latest edition of the *LAPACK Users' Guide* at the Netlib repository at the following URL:

http://www.netlib.org/lapack/lug/lapack_lug.htm

Purpose and audience

This guide describes the MLIB software library and shows how to use it. This library provides mathematical software and computational kernels for applications.

The *HP MLIB User's Guide* addresses experienced programmers who:

- Convert, develop, or optimize programs for use on HP servers and workstations
- Optimize existing software to improve performance and increase productivity

Organization

The *HP MLIB User's Guide* describes HP MLIB VECLIB in Part 1 and HP MLIB LAPACK in Part 2.

To learn fundamental information necessary for using the VECLIB library, read Chapter 1 and the introductory sections of the other chapters. These sections of background information will help you efficiently use the library subprograms.

To learn more about the subject of any chapter, refer to the literature cited in the "Associated documentation" section of each chapter.

Part 1 of this document is organized as follows:

- Chapter 1 introduces general concepts about VECLIB
- Chapter 2 describes basic vector operations included in VECLIB
- Chapter 3 describes basic matrix operations included in VECLIB
- Chapter 4 explains sparse symmetric linear equation subprograms
- Chapter 5 describes sparse symmetric eigenvalue subprograms
- Chapter 6 describes the discrete Fourier transforms in VECLIB
- Chapter 7 describes subprograms to compute convolutions and correlations of data sets
- Chapter 8 describes miscellaneous subprograms to produce random numbers, sort the elements of a vector in ascending or descending order, measure time, allocate dynamic memory, and report errors

Part 2 of this document is organized as follows:

- Chapter 9 describes information specific to Hewlett-Packard's implementation of LAPACK
- Chapter 10 describes selected LAPACK auxiliary subprograms

Organization

Supplemental material is provided as follows:

- Appendix A describes how to call VECLIB and LAPACK subprograms from within C programs
- Appendix B describes LINPACK subprograms available in HP MLIB
- Appendix C lists parallelized subprograms in VECLIB and LAPACK
- An index is included at the back of the manual

Supplemental information for Part 2 of this *HP MLIB User's Guide*, is found in the *LAPACK Users' Guide*. The latter publication from the Society for Industrial and Applied Mathematics provides an introduction to the design of LAPACK as well as complete specifications for all the driver and computational routines. You can access the latest edition of the *LAPACK Users' Guide* at the Netlib repository at the following URL:

http://www.netlib.org/lapack/lug/lapack_lug.html

Notational conventions

The following conventions are used in this manual:

Italics

Italics within text indicate mathematical entities used or manipulated by the program: for example, solve the n-by-n system of linear equations $Ax = b$.

Italics within command lines indicate generic commands, file names, or subprogram names. Substitute actual commands, file names, or subprograms for the *italicized* words. For example, the command line

f90 prog_name.o

instructs you to type the command f90, followed by the name of a program or subprogram object file.

UPPERCASE BOLDFACE

UPPERCASE BOLDFACE within text and in prototype Fortran statements indicates Fortran keywords and subprogram names that must be typed just as they appear. For example, **CALL SGESL**.

lowercase boldface

lowercase boldface within text indicates Fortran generic variable or array names. You should substitute actual variable or array names. The *italicized* mathematical entities and the **lowercase boldface** variable and array names usually correspond. For example, A is a matrix and **a** is the Fortran array containing the matrix:

CALL SGESL (a, lda, n, ipvt, b, job)

lowercase boldface within command lines indicates ASCII characters that must be typed just as they appear. For example, the command line

f90 prog_name.o

instructs you to type the command *f90*, followed by the name of a program or subprogram object file.

UPPERCASE MONOSPACE

`UPPERCASE MONOSPACE` indicates Fortran programs.

Brackets ([ ])

Square brackets in command examples designate optional entries.

NOTE

A **NOTE** highlights important supplemental information.

Documentation resources

Both the *HP MLIB User's Guide* and the *LAPACK Users' Guide* are available in hardcopy and online formats.

For the *HP MLIB User's Guide*, refer to:
http:// www.hp.com/go/mlib

For the latest edition of the *LAPACK Users' Guide*, refer to:
http://www.netlib.org/lapack/lug/lapack_lug.html

The following documents provide supplemental information:

- *LAPACK Users' Guide* Philadelphia, PA: Society for Industrial and Applied Mathematics, 1995. This guide provides information on the subprograms provided with the LAPACK library.
- *Parallel Programming Guide for HP-UX Systems*. Describes efficient shared-memory parallel programming techniques using HP compilers.
- *Fortran / 9000 Programmer's Reference*. Provides language reference information for HP Fortran 77.
- *Fortran / 9000 Programmer's Guide*. Describes features and requirements in terms of the tasks a programmer might perform. These tasks include how to compile, link, run, debug, and optimize programs.
- *HP-UX Floating-Point Guide*. Describes how floating-point arithmetic is implemented on HP 9000 systems and discusses how floating-point behavior affects the programmer.

- *HP Fortran 90 Programmer's Guide*. Provides extensive usage information (including how to compile and link), suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines for HP Fortran 90.
- *HP Fortran 90 Programmer's Reference*. Provides complete Fortran 90 language reference information. It also covers compiler options, compiler directives, and library information.
- *HP Fortran 90 Programmer's Guide*. Provides usage information (including how to compile and link), suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines from HP Fortran 90.
- *HP C Programming Guide*. Contains detailed discussions of selected C topics.
- *HP C/HP-UX Reference Manual*. Presents reference information on the C programming language as implemented by HP.
- *HP aC++ Online Programmer's Guide*. Presents reference and tutorial information on aC++. (This manual is accessed by specifying `aCC` with the `+help` command-line option.)
- *CXperf User's Guide*. Describes how to use the HP CXperf performance analyzer.
- *CXperf Command Reference*. Provides detailed description of CXperf command syntax and usage.
- *CXperf Online Help*. Product-accessible guide that explains the operations of CXperf and the steps needed to create and interpret a CXperf profile.
- *HP MPI User's Guide*. Describes how to use HP MPI (Message Passing Interface), a library of C- and Fortran-callable routines used for message-passing programming.
- *HP-UX Assembly Language Reference Manual*. Describes the HP-UX assembler for the PA-RISC processor.
- *HP PA-RISC 2.0 Architecture Reference*. Describes the architecture of the PA-RISC 2.0 processor.
- *PA-RISC Procedure Calling Conventions Reference*. Describes the conventions for creating PA-RISC assembly language procedure calls.

Documentation resources

Part 1

HP VECLIB

1 Introduction to VECLIB

Overview

VECLIB, a component of HP MLIB, is a collection of subprograms optimized for use on Hewlett-Packard servers and workstations, providing mathematical software and computational kernels for engineering and scientific applications. This library contains subprograms for:

- Dense vector operations, including the Basic Linear Algebra Subprograms (Level 1, 2, and 3 BLAS as well as BLAS Standard)
- Sparse vector operations, including the Sparse BLAS
- Matrix operations, including the Level 2 BLAS, Level 3 BLAS, and BLAS Standard
- Sparse symmetric and structurally-symmetric linear equation solutions
- Sparse symmetric ordinary and generalized eigensystem solutions
- Discrete Fourier transforms
- Convolution and correlation
- Miscellaneous tasks, such as sorting and generating random numbers

Although VECLIB was designed for use with Fortran programs, C programs can call VECLIB subprograms. Refer to Appendix A, "Calling MLIB routines from C," for details. Examples are in Fortran, unless otherwise indicated.

Except for subprograms described in Appendix B, "LINPACK Subprograms," LINPACK, EISPACK, and SKYLINE subprograms are not included in the Hewlett-Packard scientific libraries. Refer to the Appendix, "Converting from LINPACK or EISPACK" in the *LAPACK Users' Guide*, for assistance converting programs that currently call LINPACK or EISPACK routines to call LAPACK or VECLIB routines instead.

This chapter provides information necessary for efficient use of VECLIB and includes discussions of:

- Standardization
- Accessing VECLIB
- Optimization
- Parallel processing
- Profiling VECLIB applications
- Roundoff effects
- Data types and precision
- VECLIB naming convention
- Data type and byte length
- Operator arguments
- Error handling
- HP MLIB man pages
- Troubleshooting

Chapter objectives

After reading this chapter you will:

- Know how to access VECLIB library subprograms
- Understand how VECLIB works in a parallel computing environment
- Know how VECLIB interacts with the CXperf performance tool
- Understand VECLIB naming conventions
- Understand roundoff effects
- Understand how VECLIB handles errors
- Know how to access the online *HP MLIB* man pages
- Know what to do if you experience trouble using VECLIB subprograms

Standardization

VECLIB conforms to a variety of existing standards. For example, it includes Basic Linear Algebra Subprograms (BLAS), levels 1, 2, and 3, and Sparse BLAS. Boeing Computer Services' VectorPak subroutine library has also served as a model for VECLIB subprograms.

These products are available with standardized user interfaces on computers ranging from microcomputers to supercomputers. Because VECLIB conforms to these standards, it is a software bridge from other computers to Hewlett-Packard servers and workstations. However, even though the user interface is standardized, internal workings of HP VECLIB subprograms have been specialized for supported computers.

HP MLIB 7.0 supports a subset of the BLAS Standard routines. BLAS standardization efforts, supported by software and hardware vendors and university groups, began with a BLAS Technical (BLAST) Forum meeting in November 1995 at the University of Tennessee. The efforts of the BLAST Forum resulted in a BLAS Standard specification in 1999. Refer to Chapter 2, "Basic Vector Operations," and Chapter 3, "Basic Matrix Operations," for details of supported subprograms.

Accessing VECLIB

The VECLIB library is available as an archive and a shared library. It consists of compiled subprograms ready for you to incorporate into your programs with the linker. Include the appropriate declarations and CALL statements in your Fortran source program and specify that VECLIB be used as an object library at link time. Refer to "Accessing LAPACK" on page 531 for details about accessing the LAPACK library.

MLIB libraries are installed in the /opt/mlib/ directory. The entire path depends on your system type, as follows:

System Type	CPU Type	HP-UX Version	Address Width	Installation Directory
C-, J-, D-, K-, L-, R-, V- or N-Class	PA-8x00	11.0 or later	32-bit	/opt/mlib/lib/pa2.0
			64-bit	/opt/mlib/lib/pa20_64

The file name of the archive VECLIB library is libveclib.a. The VECLIB shared library is libveclib.sl.

Performance of your applications is better when you use archive libraries. However, if you need to keep executable files to a minimum size, you can use shared libraries on any pa2.0 system running the HP-UX 11.0 or later operating system. See "Linking with libisamstub.a" on page 6 if your application calls Fortran77 routines from C language code, and you are linking with the VECLIB archive library.

NOTE The HP VECLIB library contains parallelized subprograms. Refer to Appendix C, "Parallelized Subprograms," for details. If your program is run on an HP-UX 11.0 system with multiple processors, see "Parallel processing" on page 7 for additional information about linking your program.

Compiling and linking

There are several ways to link your program with the version of VECLIB tuned for the machine on which the program runs. By default, the **-lveclib** option on the **f77**, **f90**, **cc**, or **c89** command line that links your program selects the library that corresponds to 32-bit addressing on the machine on which you link.

f77 usage is similar to **f90** usage. However, **f77** does not support 64-bit addressing. **cc** is the HP C compiler and **c89** is the HP POSIX-conforming C compiler. The remainder of this book refers to the **cc** and **f90** compilers. **cc** and **f90** examples also apply to **c89** and **f70** compilers, respectively.

Method 1 below displays how to link a program that uses VECLIB on the same machine; methods 2 through 4 allow you to link a program for use on a specific machine type.

NOTE

When you use the **-aarchive_shared** flag on your compiler command line, it ensures that the compiler links the archive library. If the archive library is not available, then it links the shared library. Using the archive library usually results in better performance of your application. However, if for example, you need to keep executable files to a minimum size, you can select the shared library first using the **-ashared_archive** flag. If you omit **-aarchive_shared** and **-ashared_archive**, the linker defaults to linking the shared library. Method 1 has an example using both flags.

1. To link a program that uses VECLIB for use on the same machine, use one of the following commands:

   f90 [*options*] *file* ... **-Wl,-aarchive_shared -lveclib**

   cc [*options*] *file* ... **-Wl,-aarchive_shared -lveclib -lcl**

   To select the shared library use the **-ashared_archive** flag as follows:

   f90 [*options*] *file* ... **-Wl,-ashared_archive -lveclib**

   cc [*options*] *file* ... **-Wl,-ashared_archive -lveclib -lcl**

2. Specify the entire path of the library file on the compiler command line that links your program. For example, to link your program with VECLIB for use with 32-bit addressing on a PA-8x00-based system, use one of the following:

   f90 [*options*] *file* ... **/opt/mlib/lib/pa2.0/libveclib.a**

   cc [*options*] *file* ... **/opt/mlib/lib/pa2.0/libveclib.a -lcl**

   Replace **libveclib.a** with **libveclib.sl** on your compiler command line if you want to link the shared library.

3. Use the **-lveclib** option on the compiler command line that links your program, preceded by one of:

   -Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0

   -Wl,-aarchive_shared,-L/opt/mlib/lib/pa20_64

   For example, the command lines in Method 2 could be written:

   f90 [*options*] *file* ... **-Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0 -lveclib**

   cc [*options*] *file* ... **-Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0 -lveclib -lcl**

4. Set the LDOPTS environment variable to include one of:

   –aarchive_shared,–L/opt/mlib/lib/pa2.0

   –aarchive_shared,–L/opt/mlib/lib/pa20_64

   For example,

   setenv LDOPTS "–aarchive_shared,–L/opt/mlib/lib/pa2.0"

   Then use the **–lveclib** option on the compiler command line that links your program:

   f90 [*options*] *file ...* **–lveclib**

   cc [*options*] *file ...* **–lveclib –lcl**

NOTE An LDOPTS specification takes precedence over using -Wl on the compiler command line. That is, if you use the LDOPTS environment variable to specify a library path, you cannot override that specification with a -Wl option on your compiler command line.

Linking both VECLIB and LAPACK libraries

If your program uses subprograms from both VECLIB and LAPACK, specify both **–llapack** and **–lveclib** on the compiler command line. For example, using method 3 above on a PA-8x00-based machine with 32-bit addressing, link with

f90 [*options*] *file...* **–Wl,–aarchive_shared,–L/opt/mlib/lib/pa2.0 –lveclib –llapack**

Each of the VECLIB and LAPACK library files is complete in itself, meaning that you do not need to link one library because you have called subprograms from another. This is because various subprograms are included in both libraries. For example, VECLIB subroutine SGEMV is called by several LAPACK subprograms, and therefore, is included in the LAPACK library. Thus, in general, you have to link only the library or libraries you need.

Linking with libisamstub.a

C language codes that call Fortran77 routines from the BLAS Standard, the sparse linear equation system, or the sparse eigenvalue system, must explicitly link the ISAM (Indexed Sequential Access Method) stubs library into the program. For example,

cc [*options*] *file ...* **–Wl,–aarchive_shared,–L/opt/fortran/lib/libisamstub.a –lveclib –lcl**

This only applies if you are linking with the VECLIB archive library.

Optimization

The key computational kernels in VECLIB have been optimized to take full advantage of your PA-RISC computer's tightly integrated architecture. Optimizations include:

- Instruction scheduling to maximize the number of machine instructions executed per clock cycle
- Algorithm restructuring to increase the number of computations per memory access
- Cache management to decrease the number of data cache misses

Refer to "Accessing VECLIB" on page 4 for instructions on linking the desired library with your program.

Parallel processing

Parallel processing is available on Hewlett-Packard K-, V-, and N-Class servers running the HP-UX 11.0 or greater operating system. These systems can divide a single computational process into small streams of execution, called *threads*. The result is that you can have more than one processor executing on behalf of the same process. Also, refer to Appendix C, "Parallelized Subprograms," for a list of HP VECLIB parallelized subprograms.

You can enable or disable parallel processing at link time or at run time. A program does not use parallelism in VECLIB unless parallel processing is enabled at both link time and at runtime.

Linking for parallel or non parallel processing

To enable parallel processing at link time, your link step must produce a multithreaded executable. Use the **+O3** and **+Oparallel** compiler options to get a multithreaded executable when you link with the Fortran or C compilers:

f90 [*options* including **+O3 +Oparallel**] *file* ... **-Wl,-aarchive_shared -lveclib**

cc [*options* including **+O3 +Oparallel**] *file* ... **-Wl,-aarchive_shared -lveclib -lcl**

To disable VECLIB's automatic parallelism at link time, omit the **+Oparallel** option:

f90 [*options*] *file* ... **-Wl,-aarchive_shared -lveclib**

cc [*options*] *file* ... **-Wl,-aarchive_shared -lveclib -lcl**

Controlling VECLIB parallelism at runtime

When you enable parallelism at link time, three methods are available at runtime to specify the extent of parallel processing in MLIB.

- Use MLIB_NUMBER_OF_THREADS, a shell environment variable that allows you to enable parallelism within MLIB subprograms and to specify the maximum number of threads that can be used in parallel regions.

  Not setting MLIB_NUMBER_OF_THREADS has the same result as setting it to 1; that is, parallel processing is disabled within MLIB subroutines. Setting MLIB_NUMBER_OF_THREADS to the number of CPUs in the system, or greater, allows parallelized MLIB subprograms to use as many CPUs as are available to the process.

  The following command lines show the C shell syntax and Korn shell syntax to use when setting the variable to eight processors:

  For C shell:

  setenv MLIB_NUMBER_OF_THREADS 8

  For Korn shell:

  export MLIB_NUMBER_OF_THREADS=8

  MLIB_NUMBER_OF_THREADS is examined on the first call to a parallelized VECLIB subprogram to establish the default parallel action within VECLIB.

  Use MLIB_SETNUMTHREADS to restore VECLIB parallel processing to its run-time default that was specified by MLIB_NUMBER_OF_THREADS. Refer to the mlib_setnumthreads(3m) man page for usage information.

- Use the subroutine MLIB_SETNUMTHREADS.

  You can call this subroutine at any time to set the maximum number of parallel threads used in subsequent VECLIB or LAPACK calls. The specified value overrides the absence of the MLIB_NUMBER_OF_THREADS environment variable or any value assigned to it.

- Use the environment variable MP_NUMBER_OF_THREADS at run time to control parallelism.

These controls set the maximum amount of parallelism that your program can use, and the VECLIB-specific mechanisms offer finer control within that maximum. Refer to "Performance benefits" below for more information.

Performance benefits

If VECLIB parallelism is enabled, each parallelized VECLIB subprogram determines at run time if multiple processors are available. If multiple processors are available, VECLIB detects whether the program is already using multiple threads, and uses this information to automatically choose between a single- or parallel-processor algorithm.

If you are using an HP server with multiple processors, you can realize the performance benefits of parallel processing in three ways:

- Call any parallelized VECLIB subprogram. Let it use parallelism internally if it determines that it is appropriate to do so based on such factors as problem size, system configuration, and user environment.
- Call VECLIB subprograms in a parallelized loop or region. To use this mechanism, you must be familiar with the techniques of parallel processing. Refer to the *Parallel Programming Guide for HP-UX Systems* for details.
- Use the Message Passing Interface (MPI) explicit parallel model. Refer to the *HP MPI User's Guide* or the MPI(1) man page for details.

VECLIB subprograms are reentrant, meaning that they may be called several times in parallel to do independent computations without one call interfering with another. You can use this feature to call VECLIB subprograms in a parallelized loop or region. The compiler does not automatically parallelize loops containing a function reference or subroutine call. You can force it to parallelize such a loop by inserting compiler directives before the loop.

For example, the following Fortran code makes parallel calls to subprogram SAXPY:

```
C$DIR LOOP_PRIVATE (J)
C$DIR LOOP_PARALLEL
      DO 10 J=1, N
         CALL SAXPY (N-I,A(I,J),A(I+1,I),1,A(I+1,J),1)
   10 CONTINUE
```

While optimizing a parallel program, you may want to make parallel calls to a VECLIB subprogram to execute independent operations where the call statements are not in a loop. The Fortran compiler does not automatically parallelize code outside a loop, but you can use the BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives to tell the compiler to parallelize such code.

If a parallelized VECLIB subprogram is called from a parallelized loop or region, the internal parallelism is disabled.

Profiling VECLIB applications

CXperf is an interactive runtime performance analysis tool for programs compiled with HP ANSI C (cc), Fortran 90 (f90), and ANSI C++ (aCC) compilers. Using CXperf, you can study the performance of a program for optimizing, benchmarking, and debugging. To use CXperf, you must first compile your program with either the **+pa** or **+pal** compiler option. These options instrument the compiled program to collect performance metrics for routines, loops, and compiler-generated parallel loops.

VECLIB is not instrumented for use with CXperf, so CXperf does not automatically collect or analyze performance information for VECLIB subroutines. However, you can use CXoi, the PA-RISC Object and Archive File Instrumenter, to insert CXperf instrumentation into the VECLIB libraries. When you use these instrumented libraries, the performance of VECLIB subprograms can be included in a CXperf analysis. See the cxoi(1) man page for more information.

CXperf is an optional product. For more information about CXperf, refer to the *CXperf User's Guide* and *CXperf Command Reference* or contact your Hewlett-Packard sales representative.

Roundoff effects

VECLIB subprograms may use a different arithmetic order of evaluation than other implementations. Different roundoff characteristics may result. Accuracy of results is usually similar to other implementations, so using VECLIB should not affect the accumulation of roundoff errors in a complete application. If it does, examine the mathematical analysis of the problem to determine if it is ill-conditioned. Ill-conditioned means that the small roundoff errors that are inadvertently introduced into any computation are magnified out of proportion to the desired result. Similarly, if results with and without VECLIB differ materially, both sets of answers are probably inaccurate and you should investigate further. If the program correctly applies stable computational algorithms, the problem itself is probably ill-posed.

Data types and precision

In general, VECLIB provides the same range of functionality for both real and complex data. For most computations, there are matching subprograms for real and complex data, but there are a few exceptions.

For example, corresponding to the subprograms for real dot products, there are subprograms for complex dot products in both the conjugated and unconjugated forms because both types of complex dot products occur. However, there is no complex analogue of the subprograms for solving a real symmetric sparse linear system.

Matching subprograms for real and complex data have been coded to maintain a close correspondence between the two. However, in some areas, the correspondence is necessarily weaker, and this has not been possible.

Most subprograms in VECLIB are provided in both 32-bit and 64-bit precision versions.

VECLIB naming convention

The name of each VECLIB subprogram is a coded specification of its function within the limits of standard Fortran 77 6-character names. Usually, the first character of a subprogram name, denoted by T, shows the predominant data type according to Table 1-1:

Table 1-1 VECLIB Naming Convention—Data Type

T	Data Type	VECLIB
S	Single Precision	REAL*4
D	Double Precision	REAL*8
I	Integer	INTEGER*4
C	Complex	COMPLEX*8
Z	Double Complex	COMPLEX*16

Basic Linear Algebra Subprograms, that is, level 1, 2, and 3 BLAS, as well as the Sparse BLAS use this naming convention. For example, subprograms that compute the sum of vector elements are named according to data type: SSUM, DSUM, ISUM, CSUM, and ZSUM. Some function subprograms use two of these letters, the first describing the data type of the function and the second indicating the type of data on which it operates.

BLAS Standard subprograms use a similar convention, with *F_* prepended to each routine name.

For example, the legacy BLAS single-precision, triangular-solve routine is named STRSM and its BLAS Standard counterpart is named F_STRSM. BLAS Standard subprograms that compute the sum of vector elements are named F_SSUM, F_DSUM, F_CSUM, and Z_SUM. Refer to "Legacy BLAS routines" on page 159 and "BLAS Standard routines" on page 259 for more information.

Data type and byte length

There is a relationship between the data type of a subprogram, designated by the first character of its name (refer to T in Table 1-1), and the byte lengths of its arguments. This relationship is shown in Table 1-2:

Table 1-2 VECLIB Argument Lengths

T	VECLIB Argument Lengths		
	INTEGER LOGICAL	REAL	COMPLEX
S	4	4	8
D	4	8	16
C	4	4	8
Z	4	8	16

Operator arguments

Some BLAS routines take input-only arguments called operators that allow for the specification of multiple related operations to be performed by a single function. Operator arguments used by the BLAS Standard routines are **NORM, SORT, SIDE, UPLO, TRANS, CONJ, DIAG**, and **JROT**. Their meanings are defined as follows:

norm Used by routines computing the norm of a vector or matrix. There are seven valid values that specify the norm to be computed, namely one-norm, real one-norm, infinity-norm and real infinity-norm for vectors and matrices, two-norm for vectors, and Frobenius-norm, max-norm, and real max-norm for matrices.

sort Used by sorting routines.There are two valid values that specify whether the data should be specified in increasing or decreasing order.

side Used by functions computing the product of two matrices A and B. There are two valid values that specify whether $A*B$ or $B*A$ should be computed.

uplo Refers to triangular matrices. There are two valid values to specify whether a matrix is upper or lower triangular.

trans Used by routines applying a matrix, say A, to another vector or another matrix. There are three valid values to specify whether the matrix (A), its transpose (A^T), or its conjugate transpose (A^*) should be applied. op(A) refers to A, A^T, or A^* depending on the input value of the **trans** operator argument. Some BLAS routines have more than one **trans** operator. For example, a general matrix multiply operation can be specified as

$$C \leftarrow op(A)op(B)$$

where A, B, and C are general matrices. A **trans** argument is needed for each of the input matrices A and B. These arguments are denoted **transA** and **transB**.

conj Used by complex routines operating with x or $\bar{x}$.

diag Refers to triangular matrices. Two values are valid to specify whether the triangular matrix has unit-diagonal or not.

jrot Used by the routine to generate Jacobi rotations. There are three valid values to specify whether the rotation is an inner rotation, an outer rotation, or a sorted rotation.

For BLAS Standard routines, specify an operator argument with a named constant value. Table 1-3 on page 15 lists the operator arguments and their associated named constants.

The actual numeric value assigned to the named constant is defined in the appropriate language include file. For example, the f77blas.h include file defines assigned values for Fortrran 77.

Table 1-3 lists the operator arguments and their associated named constants.

Table 1-3 BLAS Standard Operator Arguments

Operator Argument	Named Constant	Meaning
norm	blas_one_norm	1-norm
	blas_real_one_norm	real 1-norm
	blas_two_norm	2-norm
	blas_frobenius_norm	Frobenius-norm
	blas_inf_norm	infinity-norm
	blas_real_inf_norm	real infinity-norm
	blas_max_norm	max-norm
	blas_real_max_norm	real-norm
sort	blas_increasing_order	sort in increasing order
	blas_decreasing_order	sort in decreasing order
side	blas_left_side	operate on the left hand side
	blas_right_side	operate on the right hand side
uplo	blas_upper	reference upper triangle only
	blas_lower	reference lower triangle only
transx	blas_trans	operate with x^T
	blas_no_trans	operate with x
	blas_conj_trans	operate with x^*
conj	blas_conj	operate with $\bar{x}$
	blas_no_conj	operate with x
diag	blas_non_unit_diag	non-unit triangular
	blas_unit_diag	unit triangular
jrot	blas_jrot_inner	inner rotation $c \geq \frac{1}{\sqrt{2}}$
	blas_jrot_outer	outer rotation $0 \leq c \leq \frac{1}{\sqrt{2}}$
	blas_jrot_sorted	sorted rotation $abs(a) \geq abs(b)$

Error handling

VECLIB subprograms are divided into two classes according to the way they detect and report usage errors:

- Low-level subprograms
- High-level subprograms

Low-level subprograms

Low-level subprograms are only minimally capable of detecting or handling errors. These subprograms attempt to do what is reasonable when a usage error occurs, but they do not warn you that something is wrong. For example, SDOT, which computes the dot product of two dense real vectors of length N, gives the mathematically proper result, zero, when $N \leq 0$. This is considered mathematically correct because, by definition, an empty sum is zero. Because SDOT conforms to the published BLAS argument list and usage, however, SDOT does not notify you that you may have made a mistake. Issuing a warning would add severe usage conflicts with codes that already use the BLAS.

Because these low-level subprograms do what is reasonable, you can simplify a program and perhaps speed it up by using VECLIB subprograms without checking for special cases. For example, relying on SDOT having a zero result for $N \leq 0$ may eliminate an IF statement that would take the same branch almost every time through a loop.

High-level subprograms

High-level subprograms detect and report errors. These subprograms usually do more work than the low-level subprograms, so the relative expense of checking and reporting errors may be small. Some possible errors are:

- Argument value errors, such as negative matrix order
- Computational errors, such as a singular matrix
- Computational problems, such as ill-conditioning

When a high-level subprogram detects an error, it responds with a success/error code, usually with the output argument **ier**.

The convention used for **ier** is:

- **ier** = 0: Successful exit
- **ier** < 0: Invalid value of an argument—computation not completed
- **ier** > 0: Failure during the computation

Some VECLIB subprograms do not have a success/error code in their argument lists, but instead call another VECLIB subprogram to process the error condition. MLIB provides the following error handlers:

- XERBLA
- XERVEC
- F_BLASERROR

Refer to the documentation for individual VECLIB subprogram to determine if one of these error handlers is used. For example, all BLAS Standard subprograms (those subprograms whose names begin with *F_*) use F_BLASERROR.

The standard versions of XERBLA, XERVEC and F_BLASERROR write an error message onto the standard error file. Execution is then terminated with a nonzero exit status. You can supply a version of the error handler that alters this action. Refer to "XERBLA" on page 258, "XERVEC" on page 526, and "F_BLASERROR" on page 511 for more information about these routines.

Routine-specific error conditions are listed in the respective subprogram documentation.

Troubleshooting

The following are suggestions to help you find the cause of a problem:

1. Verify that the subprogram usage in the program matches the subprogram specifications in the documentation. Pay attention to the number of arguments in the **CALL** statement and to the declarations of arrays and integer constants or variables that describe them. Write out all the arguments immediately before and after the **CALL** statement.

2. Make sure there really is a problem. For example, if you get an apparently incorrect answer, check if the answer satisfies the problem as defined in the program. For problems with more than one answer, VECLIB may produce a different answer or give the answers in a different order than expected. If the problem is ill-conditioned, VECLIB may not be able to compute a reliable answer. Error messages often suggest the cause of the problem.
3. Isolate the problem. If possible, write a small test program that encounters the same difficulty. For example, write the data causing the problem from the original program into the small one. In this way, you eliminate extraneous code from suspicion. If the problem area is large, try to pare it to a manageable size. For example, if a 50-by-50 linear system fails, try to produce a 2-by-2 system that fails in the same way.

HP MLIB man pages

The HP MLIB man pages contain online documentation that includes information from the *HP MLIB User's Guide*.

The HP MLIB man pages are installed in the directory /opt/mlib/share/man. Set this path in your MANPATH environment variable to access man pages for VECLIB and LAPACK.

HP VECLIB man pages provide:

- An introduction to VECLIB (veclib (3m))
- An introduction to each group of subprograms, for example, blas2(3m)
- A man page for each subprogram

The *HP MLIB User's Guide* has more detailed information than the man pages because of the limited number of fonts supported and the difficulty of presenting mathematical equations in the man(1) system.

For further explanation and a table of contents of reference entries for VECLIB, refer to the veclib(3) man page.

2 Basic Vector Operations

Overview

This chapter explains how to use the VECLIB vector subprograms that serve as building blocks for many user programs. It describes subprograms for performing dense and sparse vector operations. This set of VECLIB subprograms includes:

- Basic Linear Algebra Subprograms
- Sparse BLAS
- Hewlett-Packard extensions to the BLAS
- BLAS Standard

The term BLAS, as used in this section, refers to all of the above listed BLAS and the Hewlett-Packard extensions to the BLAS.

BLAS standardization efforts, supported by software and hardware vendors and university groups, began with a BLAS Technical (BLAST) Forum meeting in November 1995 at the University of Tennessee. The efforts of the BLAST Forum resulted in a BLAS Standard specification in 1999.

HP MLIB 7.0 supports a subset of the BLAS Standard routines. This chapter describes dense and banded BLAS Standard functionality in "BLAS Standard routines" on page 117.

This section discusses commonly used or computationally expensive operations of linear algebra. Even though you can code most of these operations in fewer than 10 lines of Fortran or C, using VECLIB subprograms can improve program performance as well as program modularity and readability. However, in some situations, you can achieve better computational performance by entering Fortran or C code than by calling one of these subprograms.

Chapter objectives

After reading this chapter you will:

- Understand BLAS storage conventions
- Know how to specify array sections
- Know how to handle backward storage
- Know how to use increment (also called stride) arguments
- Understand the vector subprograms included with HP VECLIB, both Legacy BLAS and BLAS Standard subprograms

Associated documentation

The following documents provide supplemental material for this chapter:

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.

Dodson, D.S., R.G. Grimes, and J.G. Lewis. "Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. June, 1991. Vol. 17, No. 2.

Lawson, C., R. Hanson, D. Kincaid, and F. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." *ACM Transactions on Mathematical Software*. September, 1979. Vol. 5, No. 3.

What you need to know to use vector subprograms

The following sections describe overall considerations for using vector subprograms:

- BLAS storage conventions
  - Fortran storage of arrays
  - Fortran array argument association
- BLAS indexing conventions
  - Forward storage
  - Backward storage
  - Increment arguments
- Operator arguments in the BLAS Standard
- Representation of a permutation matrix
- Representation of a Householder matrix

BLAS storage conventions

The Basic Linear Algebra Subprograms (BLAS) were developed to enhance the portability of published linear algebra codes. In particular LAPACK, the high-level public-domain linear equation package, uses the BLAS. When your routines use the VECLIB BLAS, you increase the efficiency of those routines on your Hewlett-Packard servers.

You need not limit your use of the VECLIB BLAS to linear algebra codes. Because these subprograms are portable, modular, and efficient, you can incorporate them into your programs. To realize the full power of the BLAS, you should understand storage and indexing conventions.

Fortran storage of arrays

Two-dimensional arrays in Fortran are stored by columns. Consider the following specifications:

```
DIMENSION A(N1,N2),B(N3)
EQUIVALENCE (A,B)
```

where N3 = N1 X N2. Then A(I,J) is associated with the same memory location as B(K) where

$K = I + (J-1) \times N1$

Successive elements of a column of A are adjacent in memory, while successive elements of a row of A are stored with a difference of N1 storage units between them. Remember that the size of a storage unit depends on the data type.

Fortran array argument association

When a Fortran subprogram is called with an array element as an argument, the value is not passed. Instead, the subprogram receives the address in memory of the element. Consider the following code segment:

```
REAL A(10,10)
J = 3
L = 10
CALL SUBR (A(1,J),L)
      .
      .
      .
SUBROUTINE SUBR (X,N)
REAL X(N)
      .
      .
      .
```

SUBR is given the address of the first element of the third column of A. Because it treats that argument as a one-dimensional array, successive elements X(1), X(2),..., occupy the same memory locations as the successive elements of the third column of A, that is, A(1,3), A(2,3), ... Hence, the entire third column of A is available to the subprogram.

BLAS indexing conventions

This section describes handling stride arguments and forward and backward storage.

A vector in the BLAS is defined by three quantities:

- Vector length
- Array or starting element within an array
- Increment, sometimes called stride—defines the number of storage units between successive vector elements

Forward storage

Suppose that X is a real array. Let N be the vector length and let INCX be the increment. Suppose that a vector x with components x_i, i= 1, 2, ..., N, is stored in X. If INCX ≥ 0, then x_i is stored in X(1 + (I−1) × INCX). This is forward storage starting from X(1) with stride equal to INCX, ending with X(1 + (N−1) × INCX). Thus, if N = 4 and INCX = 2, the vector components x_1, x_2, x_3, and x_4 are stored in the array elements X(1), X(3), X(5), and X(7), respectively.

Backward storage

Some BLAS subprograms permit the backward storage of vectors, which is specified by using a negative INCX. If INCX < 0, then x_i is stored in X(1 + (N−I) × |INCX|) or equivalently in X(1 − (N−I) × INCX). This is backward storage starting from X(1 − (N−1) × INCX) with stride equal to INCX, ending with X(1). Thus, if N = 4 and INCX = −2, the vector components x_1, x_2, x_3, and x_4 are stored in the array elements X(7), X(5), X(3), and X(1), respectively.

INCX = 0 is only permitted by COPY routines in the legacy BLAS subprograms. When INCX = 0 is allowed, it means that x is a vector of length N, whose components all equal the value of X(1).

The notation (N,X,INCX) describes a BLAS vector. For example, if X is an array of dimension N, then (N,X,1) represents forward storage and (N,X,−1) represents backward storage. If A is an M-by-N array, then (M,A(1,J),1) represents column J and (N,A(I,1),M) represents row I. Finally, if an M-by-N matrix is embedded in the upper left-hand corner of an array B of size LDB by NMAX, then column J is (M,B(1,J),1) and row is I(N,B(I,1),LDB).

Increment arguments

The following examples illustrate how to use increment arguments to perform different operations with the same subprogram. These examples use the function F_SDOT with the following usage:

```
SUBROUTINE F_SDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)
INTEGER    CONJ, INCX, INCY, N
REAL*4     ALPHA, BETA, R, X(*), Y(*)
```

This usage adds the scaled dot product of the vectors X(*) and Y(*) to a scaled scalar R.

Example 1

Compute the dot product

```
R = X(1)*Y(1) + X(2)*Y(2) + X(3)*Y(3) + X(4)*Y(4):
```

```
REAL*4    ALPHA,BETA,R,X(*),Y(*)
SUBROUTINE F_SDOT (CONJ, 4, 1.0, X, 1, 0.0, Y, 1)
```

Example 2

Compute the dot product

```
T = X(1)*Y(4) + X(2)*Y(3) + X(3)*Y(2) + X(4)*Y(1):
```

```
REAL*4    ALPHA,BETA,R,X(*),Y(*)
SUBROUTINE F_SDOT (CONJ, 4, 1.0, X, 1, 0.0, Y, -1)
```

Example 3

Compute the dot product

```
Y(2) = A(2,1)*X(1) + A(2,2)*X(2) + A(2,3)*X(3)
```

This is the dot product of the second row of an M-by-3 matrix A, stored in a 10-by-3 array, with a 3-vector X:

```
PARAMETER (LDA = 10)
REAL*4 SDOT,A(LDA,3),X(3),Y(LDA)
N = 3
Y(2) = SDOT (N, A(2,1),LDA, X,1)
```

Operator arguments in the BLAS Standard

Some routines in the BLAS Standard take input-only arguments called operators. Operators allow for the specification of multiple related operations to be performed by a single function. The BLAS Standard specifies the type and the valid values these arguments should have according to the specific programming language.

Operator arguments used by the BLAS Standard routines are **NORM, SORT, SIDE, UPLO, TRANS, CONJ, DIAG,** and **JROT**. Refer to "Operator arguments" on page 13 for explanations of the valid operator values.

In BLAS Standard routines, you specify an operator argument with a named constant value. The actual numeric value assigned to the named constant is defined in the appropriate language's include file. Operator arguments are represented in the Fortran 77 interface as INTEGERs. This specification is different from the legacy BLAS, where operator arguments are defined as CHARACTER*1.

Refer to individual routines in "BLAS Standard routines" on page 117 for the named constants you can use to specify operator arguments for basic vector subprograms.

Representation of a permutation matrix

This section explains how the BLAS Standard represents a permutation matrix.

An n-by-n permutation matrix P is represented as a product of at most n interchange permutations. An interchange permutation E is a permutation obtained by swapping two rows of the identity matrix. An efficient way to represent a general permutation matrix P is with an integer vector p of length n. In other words, $P = E_n \ldots E_1$ and each E_i is the identity with rows i and p_i interchanged:

For $i = n$ to 1 and $incp < 0$, $x(i) \leftrightarrow x(p(i))$

For $i = 1$ to n and $incp > 0$, $x(i) \leftrightarrow x(p(i))$

Representation of a Householder matrix

This section explains how the BLAS Standard represents a Householder matrix.

An elementary reflector (or elementary Householder matrix) H of order n is a unitary matrix of the form

$$H = I - \tau \upsilon \upsilon^H$$

where τ is a scalar, and υ is an n-vector, with

$$|\tau|^2 \|\upsilon\|_2^2 = 2Re(\tau)$$

υ is often referred to as the Householder vector. Often, υ can have leading or trailing zero elements, but for the purposes of this discussion, assume that H has no special structure.

This representation sets $\upsilon 1 = 1$, meaning that $\upsilon 1$ need not be stored. In real arithmetic, $1 \le \tau \le 2$, except that $\tau = 0$ implies $H = I$. This representation agrees with that used in LAPACK.

In complex arithmetic, τ may be complex, and satisfies

$$1 \le Re(\tau) \le 2 \text{ and } |\tau - 1| \le 1$$

Thus a complex H is not Hermitian, as with other representations, but it is unitary. The latter is the important property. The advantage of allowing τ to be complex is that, given an arbitrary complex vector x, H can be computed so that

$$H^* x = \beta(1, 0, \ldots, 0)^T$$

with real β. This is useful, for example, when reducing a complex Hermitian matrix to a real symmetric tridiagonal matrix, or a complex rectangular matrix to real bidiagonal form.

Subprograms for basic vector operations

The following sections in this chapter describe the vector subprograms included with VECLIB:

- Legacy BLAS routines
- BLAS Standard routines

Note that the specification for operator arguments is different in legacy BLAS routines than in BLAS Standard routines. Operator arguments are represented in the BLAS Standard Fortran 77 interface as INTEGERs; in the legacy BLAS they are defined as CHARACTER*1.

In BLAS Standard routines, you specify an operator argument with a named constant value. Refer to the individual routines in "BLAS Standard routines" on page 117 for the named constants you can use to specify operator arguments. The actual numeric value assigned to the named constant is defined in the f77blas.h include file.

Legacy BLAS routines

Name ISAMAX/IDAMAX/IIAMAX/ICAMAX/IZAMAX
Index of maximum of magnitudes

Purpose Given a real or integer vector x of length n, ISAMAX, IDAMAX, or IIAMAX determines the index of the element of the vector of maximum magnitude. Specifically, the subprograms determine the smallest index i such that

$$|x_i| = \max\left(|x_j| : j = 1, 2, \ldots, n\right)$$

Given a complex vector x of length n, ICAMAX or IZAMAX determines the smallest index i:

$$|Re(x_i)| + |Im(x_i)| = \max(|Re(x_j)| + |Im(x_j)| : j = 1, 2, \ldots, n)$$

where $Re(x_i)$ and $Im(x_i)$ are the real and imaginary parts of x_i, respectively. The usual definition of complex magnitude is

$$\left\{Re(x_i)^2 + Im(x_i)^2\right\}^{1/2}$$

This definition is not used because of computational speed. If the index i is used for pivot selection in matrix factorization, no significant difference in numerical stability should result.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISAMAX, n, incx
REAL*4          x(lenx)
i = ISAMAX(n, x, incx)

INTEGER*4       i, IDAMAX, n, incx
REAL*8          x(lenx)
i = IDAMAX(n, x, incx)

INTEGER*4       i, IIAMAX, n, incx, x(lenx)
i = IIAMAX(n, x, incx)

INTEGER*4       i, ICAMAX, n, incx
COMPLEX*8       x(lenx)
i = ICAMAX(n, x, incx)
```

INTEGER*4 **i, IZAMAX, n, incx**
COMPLEX*16 **x(lenx)**
i = IZAMAX(n, x, incx)

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

i If **n** ≤ 0, then **i** = 0. Otherwise, **i** is the index of the element of x of maximum magnitude.

Fortran Equivalent

```
      INTEGER*4 FUNCTION ISAMAX (N,X,INCX)
      REAL*4 X(*),TEMP,XMAX
      ISAMAX = 1
      IF ( N .GT. 1 ) THEN
          XMAX = ABS ( X(1) )
          INCXA = ABS ( INCX )
          IX = 1 + INCXA
          DO 10 I = 2, N
              TEMP = ABS ( X(IX) )
              IF ( TEMP .GT. XMAX ) THEN
                  ISAMAX = I
                  XMAX = TEMP
              END IF
              IX = IX + INCXA
10        CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
          ISAMAX = 0
      END IF
      RETURN
      END
```

Example Locate the largest element of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 I,IDAMAX,N,INCX
REAL*8    X(20)
N = 10
INCX = 1
I = IDAMAX (N,X,INCX)
```

Name

ISAMIN/IDAMIN/IIAMIN/ICAMIN/IZAMIN
Index of minimum of magnitudes

Purpose

Given a real or integer vector x of length n, ISAMIN, IDAMIN, or IIAMIN determines the index of element of the vector of minimum magnitude. Specifically, the subprograms determine the smallest index i such that

$$|x_i| = \min(|x_j| : j = 1, 2, \ldots, n)$$

Given a complex vector x of length n, ICAMIN or IZAMIN determines the smallest index i:

$$|Re(x_i)|+|Im(x_i)| = \min\left(|Re(x_j)|+|Im(x_j)| : j = 1, 2, \ldots, n\right)$$

where $Re(x_i)$ and $Im(x_i)$ are the real and imaginary parts of x_i, respectively. The usual definition of complex magnitude is

$$\left\{Re(x_i)^2 + Im(x_i)^2\right\}^{1/2}$$

This definition is not used because of computational speed.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISAMIN, n, incx
REAL*4          x(lenx)
i = ISAMIN(n, x, incx)

INTEGER*4       i, IDAMIN, n, incx
REAL*8          x(lenx)
i = IDAMIN(n, x, incx)

INTEGER*4       i, IIAMIN, n, incx, x(lenx)
i = IIAMIN(n, x, incx)

INTEGER*4       i, ICAMIN, n, incx
COMPLEX*8       x(lenx)
i = ICAMIN(n, x, incx)

INTEGER*4       i, IZAMIN, n, incx
COMPLEX*16      x(lenx)
i = IZAMIN(n, x, incx)
```

Input

n	Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.
incx	Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

i	If **n** ≤ 0, then **i** = 0. Otherwise, **i** is the index of the element of x of minimum magnitude.

Fortran Equivalent

```
      INTEGER*4 FUNCTION ISAMIN (N,X,INCX)
      REAL*4 X(*),TEMP,XMIN
      ISAMIN = 1
      IF ( N .GT. 1 ) THEN
          XMIN = ABS ( X(1) )
          INCXA = ABS ( INCX )
          IX = 1 + INCXA
          DO 10 I = 2, N
              TEMP = ABS ( X(IX) )
              IF ( TEMP .LT. XMIN ) THEN
                  ISAMIN = I
                  XMIN = TEMP
              END IF
              IX = IX + INCXA
   10     CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
          ISAMIN = 0
      END IF
      RETURN
      END
```

Example Locate the smallest element of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 I,IDAMIN,N,INCX
REAL*8    X(20)
N = 10
INCX = 1
I = IDAMIN (N,X,INCX)
```

Name ISCTxx/IDCTxx/IICTxx/ICCTxx/IZCTxx
Count selected vector elements

Purpose Given a real, integer, or complex vector x of length n, these subprograms count the number of elements of the vector that satisfy a specified relationship to a given scalar a.

The last two characters of the subprogram name specify the relation of interest between the elements of the vector and the scalar. For real and integer subprograms, these characters, represented by "xx" in the prototype Fortran statements, and the corresponding function values can be:

xx	Function value
EQ	$\#\{i : x_i = a\}$
GE	$\#\{i : x_i \geq a\}$
GT	$\#\{i : x_i > a\}$
LE	$\#\{i : x_i \leq a\}$
LT	$\#\{i : x_i < a\}$
NE	$\#\{i : x_i \neq a\}$

For complex subprograms, these characters and corresponding function values are:

xx	Function value
EQ	$\#\{i : x_i = a\}$
NE	$\#\{i : x_i \neq a\}$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISCTxx, n, incx
REAL*4          a, x(lenx)
i = ISCTxx(n, x, incx, a)

INTEGER*4       i, IDCTxx, n, incx
REAL*8          a, x(lenx)
i = IDCTxx(n, x, incx, a)

INTEGER*4       i, IICTxx, n, incx, a, x(lenx)
i = IICTxx(n, x, incx, a)
```

```
INTEGER*4       i, ICCTxx, n, incx
COMPLEX*8       a, x(lenx)
i = ICCTxx(n, x, incx, a)

INTEGER*4       i, IZCTxx, n, incx
COMPLEX*16      a, x(lenx)
i = IZCTxx(n, x, incx, a)
```

Input

n Number of elements of vector x to be compared to a. If $\mathbf{n} \leq 0$, the subprograms do not reference **x**.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment $|\mathbf{incx}|$; that is, x_i is stored in $\mathbf{x}((i-1)\times|\mathbf{incx}|+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

a The scalar a.

Output

i If $\mathbf{n} \leq 0$, then **i** = 0. Otherwise, **i** is the number of elements of x that satisfy the relationship with a specified by the subprogram name.

Fortran Equivalent

```
      INTEGER*4 FUNCTION ISCTEQ (N,X,INCX,A)
      REAL*4 A,X(*)
      ISCTEQ = 0
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          IF ( X(IX) .EQ. A ) ISCTEQ = ISCTEQ + 1
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Count the number of positive elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 I,IDCTGT,N,INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 0.0D0
I = IDCTGT (N,X,INCX,A)
```

Name

ISMAX/IDMAX/IIMAX
Index of maximum element of vector

Purpose

Given a real or integer vector x of length n, these subprograms determine the index of the maximum element of the vector. Specifically, the subprograms determine the smallest index i such that

$x_i = \max(x_j : j = 1, 2, \ldots, n)$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISMAX, n, incx
REAL*4          x(lenx)
i = ISMAX(n, x, incx)

INTEGER*4       i, IDMAX, n, incx
REAL*8          x(lenx)
i = IDMAX(n, x, incx)

INTEGER*4       i, IIMAX, n, incx, x(lenx)
i = IIMAX(n, x, incx)
```

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

i If **n** ≤ 0, then **i** = 0. Otherwise, **i** is the index of the maximum element of x.

Fortran Equivalent

```
      INTEGER*4 FUNCTION ISMAX (N,X,INCX)
      REAL*4 X(*),XMAX
      ISMAX = 1
      IF ( N .GT. 1 ) THEN
          XMAX = X(1)
          INCXA = ABS ( INCX )
          IX = 1 + INCXA
          DO 10 I = 2, N
              IF ( X(IX) .GT. XMAX ) THEN
                  ISMAX = I
                  XMAX = X(IX)
              END IF
              IX = IX + INCXA
10        CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
          ISMAX = 0
      END IF
      RETURN
      END
```

Example

Locate the largest element of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
      INTEGER*4 I,IDMAX,N,INCX
      REAL*8    X(20)
      N = 10
      INCX = 1
      I = IDMAX (N,X,INCX)
```

Name

ISMIN/IDMIN/IIMIN
Index of minimum element of vector

Purpose

Given a real or integer vector x of length n, these subprograms determine the index of minimum element of the vector. Specifically, the subprograms determine the smallest index i such that

$x_i = \min(x_j : j = 1, 2, \ldots, n)$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISMIN, n, incx
REAL*4          x(lenx)
i = ISMIN(n, x, incx)

INTEGER*4       i, IDMIN, n, incx
REAL*8          x(lenx)
i = IDMIN(n, x, incx)

INTEGER*4       i, IIMIN, n, incx, x(lenx)
i = IIMIN(n, x, incx)
```

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

i If **n** ≤ 0, then **i** = 0. Otherwise, **i** is the index of the minimum element of x.

Fortran Equivalent

```
      INTEGER*4 FUNCTION ISMIN (N,X,INCX)
      REAL*4 X(*),XMIN
      ISMIN = 1
      IF ( N .GT. 1 ) THEN
          XMIN = X(1)
          INCXA = ABS ( INCX )
          IX = 1 + INCXA
          DO 10 I = 2, N
              IF ( X(IX) .LT. XMIN ) THEN
                  ISMIN = I
                  XMIN = X(IX)
              END IF
              IX = IX + INCXA
10        CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
          ISMIN = 0
      END IF
      RETURN
      END
```

Example Locate the smallest element of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
      INTEGER*4 I,IDMIN,N,INCX
      REAL*8    X(20)
      N = 10
      INCX = 1
      I = IDMIN (N,X,INCX)
```

Name

ISSVxx/IDSVxx/IISVxx/ICSVxx/IZSVxx
Search vector for element

Purpose

Given a real, integer, or complex vector x of length n, these subprograms search sequentially through the vector for the first element x_i that satisfies a specified relationship to a given scalar a and return the index i of that element.

The last two characters of the subprogram name specify the relationship of interest between the element of the vector and the scalar. For real and integer subprograms, these characters, represented by "xx" in the prototype Fortran statements, and the corresponding function values can be:

xx	Function value
EQ	$\min\{i : x_i = a\}$
GE	$\min\{i : x_i \geq a\}$
GT	$\min\{i : x_i > a\}$
LE	$\min\{i : x_i \leq a\}$
LT	$\min\{i : x_i < a\}$
NE	$\min\{i : x_i \neq a\}$

For complex subprograms, these characters and corresponding function values are:

xx	Function value
EQ	$\min\{i : x_i = a\}$
NE	$\min\{i : x_i \neq a\}$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       i, ISSVxx, n, incx
REAL*4          a, x(lenx)
i = ISSVxx(n, x, incx, a)
```

```
INTEGER*4       i, IDSVxx, n, incx
REAL*8          a, x(lenx)
i = IDSVxx(n, x, incx, a)
```

```
INTEGER*4       i, IISVxx, n, incx, a, x(lenx)
i = IISVxx(n, x, incx, a)

INTEGER*4       i, ICSVxx, n, incx
COMPLEX*8       a, x(lenx)
i = ICSVxx(n, x, incx, a)

INTEGER*4       i, IZSVxx, n, incx
COMPLEX*16      a, x(lenx)
i = IZSVxx(n, x, incx, a)
```

Input

n	Number of elements of vector x to be compared to a. If **n** ≤ 0, the subprograms do not reference **x**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.
incx	Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.
a	The scalar a.

Output

i	If **n** ≤ 0 or if no element of x satisfies the relationship with a specified by the subprogram name, then **i** = 0. Otherwise, **i** is the index i of the first element x_i of x that satisfies the relationship with a specified by the subprogram name. Recall that x_i is stored in **x**((*i*−1)×|**incx**|+1).

Fortran Equivalent

```
      INTEGER*4 FUNCTION IISVEQ (N, X,INCX, A)
      INTEGER*4 X(*),A
      IISVEQ = 0
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          IF ( X(IX) .EQ. A ) THEN
              IISVEQ = I
              RETURN
          END IF
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Search for the first positive element of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 I,IDSVGT,N,INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 0.0D0
I = IDSVGT (N,X,INCX,A)
```

Name SAMAX/DAMAX/IAMAX/SCAMAX/DZAMAX
Maximum of magnitudes

Purpose Given a real or integer vector x of length n, SAMAX, DAMAX, or IAMAX computes the l_∞ norm of x, that is, the maximum of the magnitudes of the elements of the vector

$$s = \|x\|_\infty = \max(|x_i| : i = 1, 2, \ldots, n).$$

Given a complex vector x of length n, SCAMAX or DZAMAX computes

$$s = \max(|Re(x_i)| + |Im(x_i)| \ : \ i = 1, 2, \ldots, n).$$

where $Re(x_i)$ and $Im(x_i)$ are the real and imaginary parts of x_i, respectively.

The usual definition of the maximum of magnitudes of a complex vector is

$$t = \|x\|_\infty = \max\left\{\{Re(x_i)^2 + Im(x_i)^2\}^{1/2} : i = 1, 2, \ldots, n\right\}.$$

s is computed instead of t because, with its lack of square roots, it is faster to compute. Because $t \le s \le \sqrt{2}t$, s is often an acceptable substitute for t.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SAMAX, x(lenx)
s = SAMAX(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DAMAX, x(lenx)
s = DAMAX(n, x, incx)

INTEGER*4       n, incx, s, IAMAX, x(lenx)
s = IAMAX(n, x, incx)

INTEGER*4       n, incx
REAL*4          s, SCAMAX
COMPLEX*8       x(lenx)
s = SCAMAX(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DZAMAX
COMPLEX*16      x(lenx)
s = DZAMAX(n, x, incx)
```

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s If **n** ≤ 0, then **s** = 0. Otherwise, **s** is the maximum of the magnitudes of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SAMAX (N,X,INCX)
      REAL*4 X(*)
      SAMAX = 0.0
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          SAMAX = MAX ( SAMAX , ABS ( X(IX) ) )
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the maximum of the magnitudes of the elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DAMAX,X(20)
N = 10
INCX = 1
S = DAMAX (N,X,INCX)
```

Name

SAMIN/DAMIN/IAMIN/SCAMIN/DZAMIN
Minimum of magnitudes

Purpose

Given a real or integer vector x of length n, SAMIN, DAMIN, or IAMIN computes the minimum of the magnitudes of the elements of the vector

$$s = \min(|x_i| : i = 1, 2, \ldots, n).$$

Given a complex vector x of length n, SCAMIN or DZAMIN computes

$$s = \min(|Re(x_i)| + |Im(x_i)| : i = 1, 2, \ldots, n)$$

where $Re(x_i)$ and $Im(x_i)$ are the real and imaginary parts of x_i, respectively. The usual definition of the minimum of magnitudes of a complex vector is

$$t = \min(\{Re(x_i)^2 + Im(x_i)^2\}^{1/2} : i = 1, 2, \ldots, n).$$

s is computed instead of t because, with its lack of square roots, it is faster to compute. Because $t \leq s \leq \sqrt{2}t$, s is often an acceptable substitute for t.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SAMIN, x(lenx)
s = SAMIN(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DAMIN, x(lenx)
s = DAMIN(n, x, incx)

INTEGER*4       n, incx, s, IAMIN, x(lenx)
s = IAMIN(n, x, incx)

INTEGER*4       n, incx
REAL*4          s, SCAMIN
COMPLEX*8       x(lenx)
s = SCAMIN(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DZAMIN
COMPLEX*16      x(lenx)
s = DZAMIN(n, x, incx)
```

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s If **n** ≤ 0, then **s** = ∞, the largest representable machine number. Otherwise, **s** is the minimum of the magnitudes of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SAMIN (N,X,INCX)
      REAL*4 X(*)
      SAMIN = ∞
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          SAMIN = MIN ( SAMIN , ABS ( X(IX) ) )
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the minimum of the magnitudes of the elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DAMIN,X(20)
N = 10
INCX = 1
S = DAMIN (N,X,INCX)
```

Name SASUM/DASUM/IASUM/SCASUM/DZASUM
Sum of magnitudes

Purpose Given a real or integer vector x of length n, SASUM, DASUM, or IASUM computes the l_1 norm of x, that is, the sum of magnitudes of the elements of the vector

$$s = \|x\|_1 = \sum_{i=1}^{n} |x_i|$$

Given a complex vector x of length n, SCASUM or DZASUM computes

$$s = \sum_{i=1}^{n} |Re(x_i)| + |Im(x_i)|$$

where $Re(x_i)$ and $Im(x_i)$ are the real and imaginary parts of x_i, respectively.

The usual definition of sum of magnitudes of a complex vector is

$$t = \|X\|_2 = \sum_{i=1}^{n} \left\{ Re(x_i)^2 + Im(x_i)^2 \right\}^{1/2}$$

s is computed instead of t because, with its lack of square roots, it is faster to compute. Because $t \leq s \leq \sqrt{2}t$, s is often an acceptable substitute for t.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SASUM, x(lenx)
s = SASUM(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DASUM, x(lenx)
s = DASUM(n, x, incx)

INTEGER*4       n, incx, s, IASUM, x(lenx)
s = IASUM(n, x, incx)

INTEGER*4       n, incx
REAL*4          s, SCASUM
COMPLEX*8       x(lenx)
s = SCASUM(n, x, incx)
```

```
INTEGER*4       n, incx
REAL*8          s, DZASUM
COMPLEX*16      x(lenx)
s = DZASUM(n, x, incx)
```

Input

n Number of elements of vector x to be used in the sum of magnitudes. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s If **n** ≤ 0, then **s** = 0. Otherwise, **s** is the sum of magnitudes of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SASUM (N, X,INCX)
      REAL*4 X(*)
      SASUM = 0.0
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
          SASUM = SASUM + ABS ( X(IX) )
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the sum of magnitudes of the elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DASUM,X(20)
N = 10
INCX = 1
S = DASUM (N,X,INCX)
```

Name

SAXPY/DAXPY/CAXPY/CAXPYC/ZAXPY/ZAXPYC
Elementary vector operation

Purpose

Given a real or complex scalar a and real or complex vectors x and y of length n, these subprograms perform the elementary vector operations

$$y \leftarrow ax + y \quad \text{and} \quad y \leftarrow a\bar{x}+y$$

where $\bar{x}$ is the complex conjugate of x. The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays, and the indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          a, x(lenx), y(leny)
CALL SAXPY(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          a, x(lenx), y(leny)
CALL DAXPY(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       a, x(lenx), y(leny)
CALL CAXPY(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       a, x(lenx), y(leny)
CALL CAXPYC(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      a, x(lenx), y(leny)
CALL ZAXPY(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      a, x(lenx), y(leny)
CALL ZAXPYC(n, a, x, incx, y, incy)
```

Input

n	Number of elements of vectors x and y to be used in the elementary vector operation. If **n** ≤ 0, the subprograms do not reference **x** or **y**.
a	The scalar a.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CAXPYC and ZAXPYC and in unconjugated form by the other subprograms.

incx Increment for the array **x**:

- **incx** ≥ 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

y If **n** ≤ 0 or **a** = 0, then **y** is unchanged. Otherwise, $ax+y$ overwrites the input.

Notes

If **incx** = 0, then x_i = **x**(1) for all i.

The result is unspecified if **incy** = 0 or if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SAXPY (N, A, X,INCX, Y,INCY)
      REAL*4 X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IF ( A .EQ. 0.0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = A * X(IX) + Y(IY)
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Compute the REAL*8 elementary vector operation

$y \leftarrow 2x + y,$

where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    A,X(20),Y(20)
N = 10
A = 2.0D0
INCX = 1
INCY = 1
CALL DAXPY (N,A,X,INCX,Y,INCY)
```

Example 2 Subtract 3 times the 4th row of a 10-by-10 matrix from the 5th row. The matrix is stored in a two-dimensional array B of dimension 20-by-21.

```
INTEGER*4 N
REAL*8    A,B(20,21)
N = 10
A = -3.0D0
CALL DAXPY (N,A,B(4,1),20,B(5,1),20)
```

Name

SAXPYI/DAXPYI/CAXPYI/ZAXPYI
Sparse elementary vector operation

Purpose

Given a real or complex scalar a, a sparse vector x stored in compact form via a set of indices and a dense vector y stored in full storage form, these subprograms perform the elementary vector operation

$y \leftarrow ax + y.$

More precisely, let x be a sparse n-vector with $m \leq n$ interesting (usually nonzero) elements, and let $\{k_1, k_2, ..., k_m\}$ be the indices of these elements. All uninteresting elements of x are assumed to be zero. Let y be an ordinary n-vector. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$ then these subprograms compute

$y_{k_i} \leftarrow a\mathbf{x}_i + y_{k_i}, \qquad i = 1, 2, ..., m.$

Usage

```
INTEGER*4       m, indx(m)
REAL*4          a, x(m), y(n)
CALL SAXPYI(m, a, x, indx, y)

INTEGER*4       m, indx(m)
REAL*8          a, x(m), y(n)
CALL DAXPYI(m, a, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*8       a, x(m), y(n)
CALL CAXPYI(m, a, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*16      a, x(m), y(n)
CALL ZAXPYI(m, a, x, indx, y)
```

Input

m	Number of interesting elements of x, **m** $\leq$ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.
a	The scalar a.
x	Array of length **m** containing the interesting elements of x. **x**$(j) = x_i$ if **indx**$(j) = i$.

	indx	Array containing the indices $\{k_i\}$ of the interesting elements of x. The indices must satisfy $1 \le \mathbf{indx}(i) \le \mathbf{n} \qquad i = 1, 2, \ldots, \mathbf{m}$ and $\mathbf{indx}(i) \ne \mathbf{indx}(j) \qquad 1 \le i \ne j \le \mathbf{m},$ where **n** is the length of **y**.
	y	Array containing the elements of y, $\mathbf{y}(i) = y_i$.
Output	**y**	If $\mathbf{m} \le 0$ or $\mathbf{a} = 0$, then **y** is unchanged. Otherwise, $ax+y$ overwrites the input. Only the elements of **y** whose indices are included in **indx** are changed.

Notes The result is unspecified if any element of **indx** is out of range, if any two elements of **indx** have the same value, or if **x**, **indx**, and **y** overlap such that any element of x or any index shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SAXPYI (M, A, X,INDX, Y)
      REAL*4 A,X(*),Y(*)
      INTEGER*4 INDX(*)
      IF ( M .LE. 0 ) RETURN
      IF ( A .EQ. 0.0 ) RETURN
      DO 10 I = 1, M
          Y(INDX(I)) = A * X(I) + Y(INDX(I))
   10 CONTINUE
      RETURN
      END
```

Example Compute the REAL*8 elementary vector operation

$y \leftarrow 2x + y,$

where x is a sparse vector with interesting elements x_1, x_4, x_5, and x_9 stored in one-dimensional array X, and y is stored in a one-dimensional array Y of dimension 20.

```
INTEGER*4 M,INDX(4)
REAL*8    A,X(4),Y(20)
DATA      INDX / 1, 4, 5, 9 /
M = 4
A = 2.0D0
CALL DAXPYI (M,A,X,INDX,Y)
```

Name

SCLIP/DCLIP/ICLIP
Two sided vector clip

Purpose

Given scalars a and b and a vector x of length n, these subprograms form the vector y by the clip operation

$$y_i = \begin{cases} a & \text{if} \quad x_i \le a \\ x_i & \text{if} \quad a < x_i < b \\ b & \text{if} \quad b \le x_i \end{cases} \qquad i = 1, 2, \ldots, n$$

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          a, b, x(lenx), y(leny)
CALL SCLIP(n, a, b, x, incx, y, incy)
```

```
INTEGER*4       n, incx, incy
REAL*8          a, b, x(lenx), y(leny)
CALL DCLIP(n, a, b, x, incx, y, incy)
```

```
INTEGER*4       n, incx, incy, a, b, x(lenx), y(leny)
CALL ICLIP(n, a, b, x, incx, y, incy)
```

Input

n Number of elements of vectors x and y to be used. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

a The scalar a.

b The scalar b.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**:

- **incx** ≥ 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((i−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((i−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**$((i-1)\times$**incy**$+1)$.
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**$((i-$**n**$)\times$**incy**$+1)$.

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, then **y** is unchanged. Otherwise, y is set as specified in "Purpose."

Notes

x and **y** can be the same array if **incx** = **incy**. Otherwise, the result is unspecified if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SCLIP (N, A,B, X,INCX, Y,INCY)
      REAL*4 A,B,X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = MIN ( MAX ( X(IX) , A ) , B )
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example

Clip the REAL*8 vector x between −1 and 1 into y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    A,B,X(20),Y(20)
N = 10
INCX = 1
INCY = 1
A = -1.0D0
B =  1.0D0
CALL DCLIP (N,A,B,X,INCX,Y,INCY)
```

Name SCLIPL/DCLIPL/ICLIPL
Left sided vector clip

Purpose Given scalar a and a vector x of length n, these subprograms form the vector y by the left-sided clip operation

$$y_i = \begin{cases} a & \text{if} \quad x_i \le a \\ x_i & \text{if} \quad x_i > a \end{cases} \qquad i = 1, 2, \ldots, n.$$

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          a, x(lenx), y(leny)
CALL SCLIPL(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          a, x(lenx), y(leny)
CALL DCLIPL(n, a, x, incx, y, incy)

INTEGER*4       n, incx, incy, a, x(lenx), y(leny)
CALL ICLIPL(n, a, x, incx, y, incy)
```

Input

n Number of elements of vectors x and y to be used. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

a The scalar a.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**:

- **incx** ≥ 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

	incy	Increment for the array **y**, **incy** ≠ 0: **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**$((i-1)\times$**incy**$+1)$. **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**$((i-$**n**$)\times$**incy**$+1)$. Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.
Output	**y**	Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, then **y** is unchanged. Otherwise, y is set as specified in "Purpose."

Notes **x** and **y** can be the same array if **incx** = **incy**. Otherwise, the result is unspecified if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SCLIPL (N, A, X,INCX, Y,INCY)
      REAL*4 A,X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = MAX ( X(IX) , A )
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example Clip the REAL*8 vector x below −1 into y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    A,X(20),Y(20)
N = 10
INCX = 1
INCY = 1
A = -1.0D0
CALL DCLIPL (N,A,X,INCX,Y,INCY)
```

Name

SCLIPR/DCLIPR/ICLIPR
Right sided vector clip

Purpose

Given scalar b and a vector x of length n, these subprograms form the vector y by the right-sided clip operation

$$y_i = \begin{cases} x_i & \text{if} \quad x_i < b \\ b & \text{if} \quad x_i \geq b \end{cases} \qquad i = 1, 2, \ldots, n.$$

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays, and the indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          b, x(lenx), y(leny)
CALL SCLIPR(n, b, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          b, x(lenx), y(leny)
CALL DCLIPR(n, b, x, incx, y, incy)

INTEGER*4       n, incx, incy, b, x(lenx), y(leny)
CALL ICLIPR(n, b, x, incx, y, incy)
```

Input

n Number of elements of vectors x and y to be used. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

b The scalar b.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**:

- **incx** ≥ 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0: y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0: y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, then **y** is unchanged. Otherwise, y is set as specified in "Purpose."

Notes

x and **y** can be the same array if **incx** = **incy**. Otherwise, the result is unspecified if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SCLIPR (N, B, X,INCX, Y,INCY)
      REAL*4 B,X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = MIN ( X(IX) , B )
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example

Clip the REAL*8 vector x above 1 into y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    B,X(20),Y(20)
N = 10
INCX = 1
INCY = 1
B =  1.0D0
CALL DCLIPR (N,B,X,INCX,Y,INCY)
```

Name SCOPY/DCOPY/ICOPY/CCOPY/CCOPYC/ZCOPY/ZCOPYC
Copy vector

Purpose Given real, integer, or complex vectors x and y of length n, these subprograms perform the vector copy operations

$y \leftarrow x$ and $y \leftarrow \bar{x}$

where $\bar{x}$ is the complex conjugate of x. The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          x(lenx), y(leny)
CALL SCOPY(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          x(lenx), y(leny)
CALL DCOPY(n, x, incx, y, incy)

INTEGER*4       n, incx, incy, x(lenx), y(leny)
CALL ICOPY(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       x(lenx), y(leny)
CALL CCOPY(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       x(lenx), y(leny)
CALL CCOPYC(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      x(lenx), y(leny)
CALL ZCOPY(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      x(lenx), y(leny)
CALL ZCOPYC(n, x, incx, y, incy)
```

Input

n Number of elements of vectors x and y to be used in the copy operation. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CCOPYC and ZCOPYC and in unconjugated form by the other subprograms.

incx Increment for the array **x**:

incx ≥ 0 x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).

incx < 0 x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "Notes" for use of **incx** = 0. Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

incy Increment for the array **y**, **incy** ≠ 0:

incy > 0 y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).

incy < 0 y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, then **y** is unchanged. Otherwise, $y \leftarrow x$.

Notes

If **incx** = 0, then y_i = **x**(1) for all i. This can be used to initialize all elements of y to a constant. Refer to "Example 2" on page 60.

The result is unspecified if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SCOPY (N, X,INCX, Y,INCY)
      REAL*4 X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = X(IX)
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Copy the REAL*8 vector x into y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL DCOPY (N,X,INCX,Y,INCY)
```

Example 2 Initialize a one-dimensional array to zero.

```
INTEGER*4 N
REAL*8    Y(20)
N = 10
CALL DCOPY (N,0.0D0,0,Y,1)
```

Name

SDOT/DDOT/CDOTC/CDOTU/ZDOTC/ZDOTU
Dot product

Purpose

Given real or complex data vectors x and y of length n, these subprograms compute the dot products

$$s = \sum_{i=1}^{n} x_i y_i \quad \text{and} \quad s = \sum_{i=1}^{n} \bar{x}_i y_i$$

where $\bar{x}$ is the complex conjugate of x. The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          s, SDOT, x(lenx), y(leny)
s = SDOT(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          s, DDOT, x(lenx), y(leny)
s = DDOT(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       s, CDOTC, x(lenx), y(leny)
s = CDOTC(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       s, CDOTU, x(lenx), y(leny)
s = CDOTU(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      s, ZDOTC, x(lenx), y(leny)
s = ZDOTC(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      s, ZDOTU, x(lenx), y(leny)
s = ZDOTU(n, x, incx, y, incy)
```

Input

n	Number of elements of vectors x and y to be used in the dot product. If **n** ≤ 0, the subprograms do not reference **x** or **y**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CDOTC and ZDOTC and in unconjugated form by the other subprograms.

incx Increment for the array **x**:

incx ≥ 0 x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).

incx < 0 x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**:

incy ≥ 0 y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).

incy < 0 y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s The resulting value of the dot product. If **n** ≤ 0, then **s** = 0. Otherwise,

$$s = \sum_{i=1}^{n} x_i y_i$$

unless the subprogram name is CDOTC or ZDOTC, in which case

$$s = \sum_{i=1}^{n} \bar{x}_i y_i$$

Notes

If **incx** = 0, then x_i = **x**(1) for all i. If **incy** = 0, then y_i = **y**(1) for all i. In either of these cases, another VECLIB subprogram would be more efficient.

Fortran Equivalent

```
      REAL*4 FUNCTION SDOT (N, X,INCX, Y,INCY)
      REAL*4 X(*),Y(*)
      SDOT = 0.0
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          SDOT = SDOT + X(IX) * Y(IY)
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Compute the REAL*8 dot product

$$s = \sum_{i=1}^{10} x_i y_i$$

where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    S,DDOT,X(20),Y(20)
N = 10
INCX = 1
INCY = 1
S = DDOT (N,X,INCX,Y,INCY)
```

Example 2 Compute the REAL*8 dot product

$$s = \sum_{i=1}^{10} x_i y_i$$

where x is the 4th row of a 10-by-10 matrix stored in a two-dimensional array X of dimension 20-by-21, and y is a vector 10 elements long stored in one-dimensional array Y of dimension 20.

```
INTEGER*4 N
REAL*8    S,DDOT,X(20,21),Y(20)
N = 10
S = DDOT (N,X(4,1),20,Y,1)
```

Name

SDOTI/DDOTI/CDOTCI/CDOTUI/ZDOTCI/ZDOTUI
Sparse dot product

Purpose

Given a real or complex sparse vector x stored in compact form via an index vector and a dense vector y stored in full storage form, these subprograms compute the sparse dot products

$$s = \sum_{i=1}^{n} x_i y_i \quad \text{and} \quad s = \sum_{i=1}^{n} \bar{x}_i y_i$$

where $\bar{x}$ is the complex conjugate of x.

More precisely, let x be a sparse n-vector with $m \leq n$ interesting (usually nonzero) elements, let $\{k_1, k_2, \ldots, k_m\}$ be the indices of these elements. (While some interesting elements of x can be zero, all uninteresting elements are assumed to be zero.) Let y be an ordinary n-vector. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$, then these subprograms compute

$$s = \sum_{i=1}^{m} x_i y_{k_i} \quad \text{and} \quad s = \sum_{i=1}^{m} \bar{x}_i y_{k_i}$$

Usage

```
INTEGER*4       m, indx(m)
REAL*4          s, SDOTI, x(m), y(n)
s = SDOTI(m, x, indx, y)

INTEGER*4       m, indx(m)
REAL*8          s, DDOTI, x(m), y(n)
s = DDOTI(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*8       s, CDOTCI, x(m), y(n)
s = CDOTCI(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*8       s, CDOTUI, x(m), y(n)
s = CDOTUI(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*16      s, ZDOTCI, x(m), y(n)
s = ZDOTCI(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*16      s, ZDOTUI, x(m), y(n)
s = ZDOTUI(m, x, indx, y)
```

Input

m Number of interesting elements of x, $\mathbf{m} \leq \mathbf{n}$. If $\mathbf{m} \leq 0$, the subprograms do not reference **x**, **indx**, or **y**.

x Array of length **m** containing the interesting elements of x. x is used in conjugated form by CDOTCI and ZDOTCI and in unconjugated form by the other subprograms.

indx Array containing the indices $\{k_i\}$ of the interesting elements of x. The indices must satisfy
$1 \leq \mathbf{indx}(i) \leq \mathbf{n}, \quad i = 1, 2, \ldots, \mathbf{m},$
where **n** is the length of **y**.

y Array containing the elements of y, $\mathbf{y}(i) = y_i$.

Output

s The resulting value of the dot product. If $\mathbf{m} \leq 0$, then $\mathbf{s} = 0$. Otherwise,

$$s = \sum_{i=1}^{m} \mathbf{x}(i) \times y(\mathbf{indx}(i))$$

unless the subprogram name is CDOTCI or ZDOTCI, in which case

$$s = \sum_{i=1}^{m} \mathrm{CONJG}(\mathbf{x}(i)) \times \mathbf{y}(\mathbf{indx}(i))$$

Fortran Equivalent

```
      REAL*4 FUNCTION SDOTI (M, X,INDX, Y)
      REAL*4 X(*),Y(*)
      INTEGER*4 INDX(*)
      SDOTI = 0.0
      IF ( M .LE. 0 ) RETURN
      DO 10 I = 1, M
          SDOTI = SDOTI + X(I) * Y(INDX(I))
   10 CONTINUE
      RETURN
      END
```

Example Compute the REAL*8 sparse dot product

$$s = \sum_{i=1}^{10} x_i y_i,$$

where x is a sparse vector with interesting elements x_1, x_4, x_5, and x_9 stored in one-dimensional array X, and y is a vector 10 elements long stored in a one-dimensional array Y of dimension 20.

```
INTEGER*4 M,INDX(4)
REAL*8    S,DDOTI,X(4),Y(20)
DATA      INDX / 1, 4, 5, 9 /
M = 4
S = DDOTI (M,X,INDX,Y)
```

Name

SFRAC/DFRAC
Extract fractional parts

Purpose

Given a real vector x of length n, these subprograms extract the fractional portions of the elements of x and return them in a vector y.

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          x(lenx), y(leny)
CALL SFRAC(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          x(lenx), y(leny)
CALL DFRAC(n, x, incx, y, incy)
```

Input

n Number of elements of vectors x and y to be used. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**:

- **incx** ≥ 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((i−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((i−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

incy Increment for the array **y**, **incy** $\neq 0$:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((i−1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((i−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output **y** Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, **y** is unchanged. Otherwise, y_i is the fractional part of x_i.

Notes The fractional part of a number is either zero or of the same sign as the number. Thus, the fractional parts of −1.4, −1.0, 0.6, and 2.0 are −0.4, 0.0, 0.6, and 0.0, respectively. **x** and **y** can be the same array if **incx** = **incy**. Otherwise, the result is unspecified if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SFRAC (N, X,INCX, Y,INCY)
      REAL*4 X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          Y(IY) = X(IX) - AINT ( X(IX) )
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example Extract the fractional parts of the elements of the REAL*8 vector x into y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL DFRAC (N,X,INCX,Y,INCY)
```

Name

SGTHR/DGTHR/IGTHR/CGTHR/ZGTHR
Gather sparse vector

Purpose

Given a real, integer, or complex dense vector y stored in full storage form and a set of indices of *interesting* elements of y, these subprograms gather those elements into a sparse vector x stored in compact form via the set of indices.

More precisely, let $\{k_1, k_2, ..., k_m\}$ be the indices of the interesting elements. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$, then

$\mathbf{x}_i = y_{k_i}, i = 1, 2, ..., m.$

Usage

```
INTEGER*4       m, indx(m)
REAL*4          y(n), x(m)
CALL SGTHR(m, y, x, indx)

INTEGER*4       m, indx(m)
REAL*8          y(n), x(m)
CALL DGTHR(m, y, x, indx)

INTEGER*4       m, indx(m), y(n), x(m)
CALL IGTHR(m, y, x, indx)

INTEGER*4       m, indx(m)
COMPLEX*8       y(n), x(m)
CALL CGTHR(m, y, x, indx)

INTEGER*4       m, indx(m)
COMPLEX*16      y(n), x(m)
CALL ZGTHR(m, y, x, indx)
```

Input

m — Number of interesting elements, **m** ≤ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.

y — Array containing the elements of y, **y**$(i) = y_i$. Only the elements of **y** whose indices are included in **indx** are accessed.

indx — Array containing the indices $\{k_i\}$ of the interesting elements of **y**. The indices must satisfy

$1 \le \mathbf{indx}(i) \le \mathbf{n}, \quad i = 1, 2, ..., \mathbf{m},$

where **n** is the length of **y**.

Output

x — If **m** ≤ 0, then **x** is unchanged. Otherwise, the **m** interesting elements of y: **x**$(j) = y_i$ if **indx**$(j) = i$.

Notes

The result is unspecified if any element of **indx** is out of range or if **x**, **indx**, and **y** overlap such that any element of y or any index shares a memory location with any element of x.

Fortran Equivalent

```
      SUBROUTINE SGTHR (M, Y, X,INDX)
      REAL*4 X(*),Y(*)
      INTEGER*4 INDX(*)
      IF ( M .LE. 0 ) RETURN
      DO 10 I = 1, M
          X(I) = Y(INDX(I))
   10 CONTINUE
      RETURN
      END
```

Example

Gather y into x, where y is a vector with interesting elements y_1, y_4, y_5, and y_9 stored in one-dimensional array Y of dimension 20, and x is a vector stored in compact form in a one-dimensional array X.

```
INTEGER*4 M,INDX(4)
REAL*8    Y(20),X(4)
DATA      INDX / 1, 4, 5, 9 /
M = 4
CALL DGTHR (M,Y,X,INDX)
```

Name

SGTHRZ/DGTHRZ/IGTHRZ/CGTHRZ/ZGTHRZ
Gather and zero sparse vector

Purpose

Given a real, integer, or complex dense vector y stored in full storage form and a set of indices of *interesting* elements of y, these subprograms gather those elements into a sparse vector x stored in compact form via the set of indices and then reset those elements of y to zero.

More precisely, let $\{k_1, k_2, ..., k_m\}$ be the indices of the interesting elements. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$, then these subprograms simultaneously perform the operations

$$\begin{cases} x_i = y_{k_i} \\ y_{k_i} = 0 \end{cases} \qquad i = 1, 2, \ldots, m$$

If all nonzero elements of y are listed in **indx**, then these subprograms simultaneously perform the vector operations

$$\begin{cases} x \leftarrow y \\ y \leftarrow 0 \end{cases}$$

Usage

```
INTEGER*4       m, indx(m)
REAL*4          y(n), x(m)
CALL SGTHRZ(m, y, x, indx)

INTEGER*4       m, indx(m)
REAL*8          y(n), x(m)
CALL DGTHRZ(m, y, x, indx)

INTEGER*4       m, indx(m), y(n), x(m)
CALL IGTHRZ(m, y, x, indx)

INTEGER*4       m, indx(m)
COMPLEX*8       y(n), x(m)
CALL CGTHRZ(m, y, x, indx)

INTEGER*4       m, indx(m)
COMPLEX*16      y(n), x(m)
CALL ZGTHRZ(m, y, x, indx)
```

Input

m Number of interesting elements of y, **m** $\leq$ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.

y Array containing the elements of y, $\mathbf{y}(i) = y_i$. Only the elements of **y** whose indices are included in **indx** are accessed.

indx Array containing the indices $\{k_i\}$ of the interesting elements of y. The indices must satisfy

$1 \leq \mathbf{indx}(i) \leq \mathbf{n}, i = 1, 2, \ldots, \mathbf{m}$

and

$\mathbf{indx}(i) \neq \mathbf{indx}(j), 1 \leq i \neq j \leq \mathbf{m},$

where **n** is the length of **y**.

Output

x If **m** ≤ 0, then **x** is unchanged. Otherwise, the **m** interesting elements of y: $\mathbf{x}(j) = y_i$ if $\mathbf{indx}(j) = i$.

y $\mathbf{y}(\mathbf{indx}(i)) = 0, \quad i = 1, 2, \ldots, \mathbf{m}.$

Notes

The result is unspecified if any element of **indx** is out of range, if any two elements of **indx** have the same value, or if **x**, **indx**, and **y** overlap such that any element of y or any index shares a memory location with any element of x.

Fortran Equivalent

```
      SUBROUTINE SGTHRZ (M, Y, X,INDX)
      REAL*4 X(*),Y(*)
      INTEGER*4 INDX(*)
      IF ( M .LE. 0 ) RETURN
      DO 10 I = 1, M
          X(I) = Y(INDX(I))
          Y(INDX(I)) = 0.0
   10 CONTINUE
      RETURN
      END
```

Example

Gather y into x and reset y to zero, where y is a vector with nonzero elements y_1, y_4, y_5, and y_9 stored in a one-dimensional array Y of dimension 20, and x is stored in compact form in a one-dimensional array X.

```
INTEGER*4 M,INDX(4)
REAL*8    Y(20),X(4)
DATA      INDX / 1, 4, 5, 9 /
M = 4
CALL DGTHRZ (M,Y,X,INDX)
```

Name

SLSTxx/DLSTxx/ILSTxx/CLSTxx/ZLSTxx
List selected vector elements

Purpose

Given a real, integer, or complex vector x of length n, these subprograms search sequentially through the vector and fill an array with a list of the indices i for which the elements x_i satisfy a specified relationship to a given scalar a.

The last two characters of the subprogram name specify the relationship of interest between the elements of the vector and the scalar. For real and integer subprograms, these characters, represented by "xx" in the prototype Fortran statements, and the corresponding list contents can be:

xx	List contents
EQ	$\{i : x_i = a\}$
GE	$\{i : x_i \geq a\}$
GT	$\{i : x_i > a\}$
LE	$\{i : x_i \leq a\}$
LT	$\{i : x_i < a\}$
NE	$\{i : x_i \neq a\}$

For complex subprograms, these characters and corresponding list contents are:

xx	List contents
EQ	$\{i : x_i = a\}$
NE	$\{i : x_i \neq a\}$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx, nindx, indx(n)
REAL*4          a, x(lenx)
CALL SLSTxx(n, x, incx, a, nindx, indx)

INTEGER*4       n, incx, nindx, indx(n)
REAL*8          a, x(lenx)
CALL DLSTxx(n, x, incx, a, nindx, indx)

INTEGER*4       n, incx, nindx, indx(n), a, x(lenx)
CALL ILSTxx(n, x, incx, a, nindx, indx)
```

```
INTEGER*4       n, incx, nindx, indx(n)
COMPLEX*8       a, x(lenx)
CALL CLSTxx(n, x, incx, a, nindx, indx)

INTEGER*4       n, incx, nindx, indx(n)
COMPLEX*16      a, x(lenx)
CALL ZLSTxx(n, x, incx, a, nindx, indx)
```

Input

n Number of elements of vector x to be compared to a. If **n** ≤ 0, the subprograms do not reference **x** or **indx**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((i−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

a The scalar a.

Output

nindx If **n** ≤ 0, then **nindx** = 0. Otherwise, **nindx** is the number of elements of x that satisfy the relationship with a specified by the subprogram name.

indx Array filled with the list of indices i of the elements x_i of x that satisfy the relationship with a specified by the subprogram name. Only the first **nindx** elements of **indx** are changed. Recall that x_i is stored in **x**((i−1)×|**incx**|+1).

Notes

These subprograms are sometimes useful for optimizing a loop containing an **IF** statement. Refer to "Example 2" on page 75.

Fortran Equivalent

```
      SUBROUTINE SLSTEQ (N,X,INCX,A,NI,INDX)
      REAL*4 X(*),A
      INTEGER*4 INDX(*)
      INCXA = ABS ( INCX )
      IX = 1
      NI = 0
      DO 10 I = 1, N
          IF ( X(IX) .EQ. A ) THEN
              NI = NI + 1
              INDX(NI) = I
          END IF
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example 1 Build a list of the indices of the positive elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX,NINDX,INDX(20)
REAL*8    A,X(20)
N = 10
INCX = 1
A = 0.0D0
CALL DLSTGT (N,X,INCX,A,NINDX,INDX)
```

Example 2 Optimize the following program segment, where the **THEN** clause of the **IF** statement is much more likely than the **ELSE** clause.

```
INTEGER*4 I,N
REAL*8    A,B,D,DLIM,R
REAL*8    F(20000),X(20000),Y(20000),Z(20000)
A = ...
B = ...
DLIM = ...
R = ...
N = 20000
DO 10 I = 1, N
   D = SQRT( X(I)**2 + Y(I)**2 + Z(I)**2 ) - R
   IF ( D .GT. DLIM ) THEN
       F(I) = A * EXP( B * D )
   ELSE
       CALL FORCE (D,F(I))
   END IF
10 CONTINUE
```

Change D to an array and introduce array INDX to hold the indices corresponding to the **ELSE** clause. Split the body of the **DO** loop into two parts. The first part corresponds to the body of the loop before the **IF** statement and the **THEN** clause. It fully optimizes, so, even though it computes a few more exponentials than the original code, it is still considerably faster. DLSTLE is then called to determine the indices for which the **ELSE** clause must be

executed, and the second **DO** loop executes the **ELSE** clause for those indices. The resulting program segment is:

```
      INTEGER*4 I,J,N,NINDX,INDX(20000)
      REAL*8 A,B,DLIM,R
      REAL*8 D(20000),F(20000),X(20000),Y(20000),Z(20000)
      A = ...
      B = ...
      DLIM = ...
      R = ...
      N = 20000
      DO 10 I = 1, N
         D(I) = SQRT( X(I)**2 + Y(I)**2 + Z(I)**2 ) - R
         F(I) = A * EXP( B * D(I) )
   10 CONTINUE
      CALL DLSTLE (N,D,1,DLIM,NINDX,INDX)
      DO 20 J = 1, NINDX
          I = INDX(J)
          CALL FORCE (D(I),F(I))
   20 CONTINUE
```

Name

SMAX/DMAX/IMAX
Maximum of vector

Purpose

Given a real or integer vector x of length n, these subprograms compute the maximum of the elements of the vector

$s = \max(x_i : i = 1, 2, \ldots, n)$.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SMAX, x(lenx)
s = SMAX(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DMAX, x(lenx)
s = DMAX(n, x, incx)

INTEGER*4       n, incx, s, IMAX, x(lenx)
s = IMAX(n, x, incx)
```

Input

n Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s If **n** ≤ 0, then **s** = $-\infty$, the most negative representable machine number. Otherwise, **s** is the maximum of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SMAX (N,X,INCX)
      REAL*4 X(*)
      SMAX = - ∞
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          SMAX = MAX ( SMAX , X(IX) )
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the maximum of the elements of REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DMAX,X(20)
N = 10
INCX = 1
S = DMAX (N,X,INCX)
```

Name SMIN/DMIN/IMIN
Minimum of vector

Purpose Given a real or integer vector x of length n, these subprograms compute the minimum of the elements of the vector

$s = \min(x_i : i = 1, 2, \ldots, n)$.

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SMIN, x(lenx)
s = SMIN(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DMIN, x(lenx)
s = DMIN(n, x, incx)

INTEGER*4       n, incx, s, IMIN, x(lenx)
s = IMIN(n, x, incx)
```

Input

n	Number of elements of vector x to be used. If **n** ≤ 0, the subprograms do not reference **x**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.
incx	Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s	If **n** ≤ 0, then **s** = ∞, the largest representable machine number. Otherwise, **s** is the minimum of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SMIN (N,X,INCX)
      REAL*4 X(*)
      SMIN = ∞
      INCXA = ABS ( INCX )
      IX = 1
      DO 10 I = 1, N
          SMIN = MIN ( SMIN , X(IX) )
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the minimum of the elements of REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DMIN,X(20)
N = 10
INCX = 1
S = DMIN (N,X,INCX)
```

Name

SNRM2/DNRM2/SCNRM2/DZNRM2
Euclidean norm

Purpose

Given a real or complex vector x of length n, these subprograms compute the Euclidean (that is, l_2) norm of the vector

$$s = \|x\|_2 = \left\{ \sum_{i=1}^{n} |x_i|^2 \right\}^{1/2}$$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SNRM2, x(lenx)
s = SNRM2(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DNRM2, x(lenx)
s = DNRM2(n, x, incx)

INTEGER*4       n, incx
REAL*4          s, SCNRM2
COMPLEX*8       x(lenx)
s = SCNRM2(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DZNRM2
COMPLEX*16      x(lenx)
s = DZNRM2(n, x, incx)
```

Input

n Number of elements of vector x to be used in the Euclidean norm. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output **s** If **n** ≤ 0, then **s** = 0. Otherwise, **s** is the Euclidean norm of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SNRM2 ( N, X,INCX )
      INTEGER*4 INCX, INCXA, IX, N
      REAL*4 ABSXI, SCALE, SSQ, X(*)
      IF ( N .GT. 1 ) THEN
         INCXA = ABS ( INCX )
         SCALE = 0.0
         SSQ   = 1.0
         DO 10 IX = 1, 1 + (N-1)*INCXA, INCXA
            IF ( X(IX) .NE.0.0 ) THEN
               ABSXI = ABS ( X(IX) )
               IF ( SCALE .LT. ABSXI ) THEN
                   SSQ   = 1.0 + SSQ * (SCALE/ABSXI) ** 2
                   SCALE = ABSXI
               ELSE
                   SSQ   = SSQ + (ABSXI/SCALE) ** 2
               END IF
            END IF
10       CONTINUE
         SNRM2 = SCALE * SQRT ( SSQ )
      ELSE IF ( N .EQ. 1 ) THEN
         SNRM2 = ABS ( X(1) )
      ELSE
         SNRM2 = 0.0
      END IF
      RETURN
      END
```

Example Compute the Euclidean norm of the REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DNRM2,X(20)
N = 10
INCX = 1
S = DNRM2 (N,X,INCX)
```

Name

SNRSQ/DNRSQ/SCNRSQ/DZNRSQ
Euclidean norm squared

Purpose

Given a real or complex vector x of length n, these subprograms compute the square of the Euclidean (that is, l_2) norm of the vector

$$s = \|x\|_2^2 = \sum_{i=1}^{n} \|x_i\|$$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SNRSQ, x(lenx)
s = SNRSQ(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DNRSQ, x(lenx)
s = DNRSQ(n, x, incx)

INTEGER*4       n, incx
REAL*4          s, SCNRSQ
COMPLEX*8       x(lenx)
s = SCNRSQ(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DZNRSQ
COMPLEX*16      x(lenx)
s = DZNRSQ(n, x, incx)
```

Input

n Number of elements of vector x to be used in the calculation. If **n** ≤ 0, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output **s** If $\mathbf{n} \leq 0$, then $\mathbf{s} = 0$. Otherwise, **s** is the square of the Euclidean norm of x.

Fortran Equivalent

```
REAL*4 FUNCTION SNRSQ (N, X,INCX)
REAL*4 X(*)
SNRSQ = 0.0
IF ( N .LE. 0 ) RETURN
IX = 1
INCXA = ABS ( INCX )
DO 10 I = 1, N
    SNRSQ = SNRSQ + X(IX) ** 2
    IX = IX + INCXA
10 CONTINUE
RETURN
END
```

Example Compute the square of the Euclidean norm of the REAL*8 vector x, where $x$is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,DNRSQ,X(20)
N = 10
INCX = 1
S = DNRSQ (N,X,INCX)
```

Name SRAMP/DRAMP/IRAMP
Generate linear ramp

Purpose Given real or integer scalars a and h, these subprograms generate a linear ramp function

$x_i = a + (i-1)h, i = 1, 2, \ldots, n.$

x can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          a, h, x(lenx)
CALL SRAMP(n, a, h, x, incx)

INTEGER*4       n, incx
REAL*8          a, h, x(lenx)
CALL DRAMP(n, a, h, x, incx)

INTEGER*4       n, incx, a, h, x(lenx)
CALL IRAMP(n, a, h, x, incx)
```

Input

n	Number of elements of x to be generated.
a	The scalar a.
h	The scalar h.
incx	Increment for the array **x**, **incx** $\neq 0$. x is stored forward in array **x** with increment $\lvert$**incx**$\rvert$; that is, x_i is stored in **x**$((i-1)\times\lvert$**incx**$\rvert+1)$. Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

x	Array of length **lenx** = (**n**−1)×$\lvert$**incx**$\rvert$+1 containing the n-vector x. If **n** ≤ 0, then **x** is not referenced. Otherwise, the specified linear ramp function replaces the input.

Notes The result is unspecified if **incx** = 0.

Fortran Equivalent

```
      SUBROUTINE SRAMP (N, X1,DX, X,INCX)
      REAL*4 X1,DX,X(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
          X(IX) = X1 + (I-1) * DX
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Generate the linear ramp x with initial value 0 and slope $\pi/9$, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    A,H,PI,X(20)
PARAMETER ( PI = 3.14159265358979323846D0 )
N = 10
INCX = 1
A = 0.0D0
H = PI / (N-1)
CALL DRAMP (N,A,H,X,INCX)
```

Name

SROT/DROT/CROT/CSROT/ZROT/ZDROT
Apply Givens rotation

Purpose

Given a real scalar c, a real or complex scalar, s and real or complex vectors x and y of length n, these subprograms apply the Givens rotation

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad \text{for } i = 1, \ldots, n$$

where $\bar{s}$ is the complex conjugate of s; $\bar{s} = s$ if s is real. The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. The indexing through the arrays can be either forward or backward.

Usually, c and s have been determined by the companion subprogram SROTG, DROTG, CROTG, or ZROTG.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          x(lenx), y(leny), c, s
CALL SROT(n, x, incx, y, incy, c, s)

INTEGER*4       n, incx, incy
REAL*8          x(lenx), y(leny), c, s
CALL DROT(n, x, incx, y, incy, c, s)

INTEGER*4       n, incx, incy
REAL*4          c
COMPLEX*8       x(lenx), y(leny), s
CALL CROT(n, x, incx, y, incy, c, s)

INTEGER*4       n, incx, incy
REAL*4          c, s
COMPLEX*8       x(lenx), y(leny)
CALL CSROT(n, x, incx, y, incy, c, s)

INTEGER*4       n, incx, incy
REAL*8          c
COMPLEX*16      x(lenx), y(leny), s
CALL ZROT(n, x, incx, y, incy, c, s)

INTEGER*4       n, incx, incy
REAL*8          c, s
COMPLEX*16      x(lenx), y(leny)
CALL ZDROT(n, x, incx, y, incy, c, s)
```

Input

n Number of elements of vectors x and y to be used in the Givens rotation. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

- **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

c The scalar c.

s The scalar s.

Output

x and **y** If **n** ≤ 0 or if **c** = 1 and **s** = 0, then **x** and **y** are unchanged. Otherwise, the result vectors overwrite the input.

Notes

The result is unspecified if **incx** = 0 or **incy** = 0 or if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

There are no companion subprograms that construct real Givens rotations for CSROT and ZDROT.

VECLIB also contains subprograms that construct and apply modified Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use, but are more efficient.

Fortran Equivalent

```
      SUBROUTINE SROT (N, X,INCX, Y,INCY, C,S)
      REAL*4 C,S,TEMP,X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IF ( C .EQ. 1.0 .AND. S .EQ. 0.0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          TEMP = C * X(IX) + S * Y(IY)
          Y(IY) = C * Y(IY) - S * X(IX)
          X(IX) = TEMP
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Apply a Givens rotation to x and y, vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    X(20),Y(20),C,S
N = 10
INCX = 1
INCY = 1
CALL DROT (N,X,INCX,Y,INCY,C,S)
```

Example 2 Reduce 10-by-10 matrix **a** stored in two-dimensional array A of dimension 20-by-21 to upper-triangular form via Givens rotations (compare with "Example 2" on page 96 in the description of SROTM and DROTM).

```
INTEGER*4 INCA,I,J,N
REAL*8    A(20,21),C,S
INCA = 20
DO 20 I = 1, 9
     N = 10 - I
     DO 10 J = I+1, 10
          CALL DROTG (A(I,I),A(J,I),C,S)
          CALL DROT  (N,A(I,I+1),INCA,A(J,I+1),INCA,C,S)
10    CONTINUE
20 CONTINUE
```

Name SROTG/DROTG/CROTG/ZROTG
Construct Givens rotation

Purpose Given real or complex scalars a and b, these subprograms construct a Givens plane rotation matrix that annihilates b. Specifically, they determine scalars c and s such that

$$\begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \bullet \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where c is real, r and s are of the same type as a and b, and $\bar{s}$ is the complex conjugate of s.

Usually, c and s are passed to companion subprogram SROT, DROT, CROT, or ZROT to apply the Givens rotation to a pair of vectors.

SROTG and DROTG also determine a quantity z that permits the later stable reconstruction of c and s from a single quantity.

Usage

```
REAL*4          a, b, c, s
CALL SROTG(a, b, c, s)

REAL*8          a, b, c, s
CALL DROTG(a, b, c, s)

REAL*4          c
COMPLEX*8       a, b, s
CALL CROTG(a, b, c, s)

REAL*8          c
COMPLEX*16      a, b, s
CALL ZROTG(a, b, c, s)
```

Input

a	The scalar a.
b	The scalar b.

Output

a The rotated result r overwrites a.

b Not used as output by CROTG and ZROTG. In SROTG and DROTG, the reconstruction quantity z overwrites b. The reconstruction quantity z is useful if a matrix is being transformed by a sequence of Givens rotations that must be saved to be applied again. Because each z overwrites an element that has been reduced to zero, the transformations can be saved without using any additional storage. The quantities c and s can be reconstructed from z as follows:

if $|z| = 0$, set $c = 0$ and $s = 1$.
if $|z| < 0$, set $c = \sqrt{(1-z^2)}$ αvδ σ = ζ.
if $|z| > 0$, set $c = 1/z$ and $s = \sqrt{(1-c^2)}$.

c The rotation scalar c.

s The rotation scalar s.

Notes VECLIB also contains subprograms that construct and apply modified Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.

Example Construct a Givens plane rotation that rotates vectors x and y in such a way as to annihilate y_1. x and y are vectors 10 elements long stored in one-dimensional arrays `X` and `Y` of dimension 20.

```
REAL*8 X(20),Y(20),C,S
CALL DROTG (X(1),Y(1),C,S)
```

`X(1)` is the rotated result and `Y(1)` is the reconstruction quantity, so these elements should not be rotated by a subsequent call to DROT.

Name

SROTI/DROTI
Apply sparse Givens rotation

Purpose

Given real scalars c and s, a sparse vector x stored in compact form via a set of indices, and a dense vector y stored in full storage form, these subprograms apply the Givens rotation

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad \text{for } i = 1, \ldots, n.$$

More precisely, let x be a sparse n-vector with $m \leq n$ interesting (usually nonzero) elements, and let $\{k_1, k_2, \ldots, k_m\}$ be the indices of these elements. All uninteresting elements of x are assumed to be zero. Let y be an ordinary n-vector that has zero elements corresponding to the uninteresting elements of x. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$, these subprograms compute

$$\begin{bmatrix} x_i \\ y_{k_i} \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_{k_i} \end{bmatrix} \qquad \text{for } i = 1, \ldots, m.$$

Usually, c and s have been determined by the companion subprogram SROTG or DROTG.

Usage

```
INTEGER*4       m, indx(m)
REAL*4          x(m), y(n), c, s
CALL SROTI(m, x, indx, y, c, s)

INTEGER*4       m, indx(m)
REAL*8          x(m), y(n), c, s
CALL DROTI(m, x, indx, y, c, s)
```

Input

m — Number of interesting elements of x, **m** $\leq$ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.

x — Array containing the interesting elements of x. **x**$(j) = x_i$ if **indx**$(j) = i$.

	indx	Array containing the indices $\{k_i\}$ of the interesting elements of x. The indices must satisfy the following: $1 \le \mathbf{indx}(i) \le \mathbf{n}, i = 1, 2, \ldots, \mathbf{m}$ $\mathbf{indx}(i) \ne \mathbf{indx}(j), 1 \le i \ne j \le \mathbf{m}$ where **n** is the length of **y**.
	y	Array containing the elements of y, $\mathbf{y}(i) = y_i$.
	c	The scalar c.
	s	The scalar s.
Output	**x and y**	If $\mathbf{m} \le 0$ or if $c = 1$ and $s = 0$, then **x** and **y** are unchanged. Otherwise, the result vectors overwrite the input. Only the elements of **y** whose indices are included in **indx** are changed.

Notes The result is unspecified if any element of **indx** is out of range, if any two elements of **indx** have the same value, or if **x**, **indx**, and **y** overlap such that any index or any element of x or y share a memory location.

Fortran Equivalent

```
      SUBROUTINE SROTI (M, X,INDX, Y, C,S)
      REAL*4 C,S,TEMP,X(*),Y(*)
      INTEGER*4 INDX(*)
      IF ( M .LE. 0 ) RETURN
      IF ( C .EQ. 1.0 .AND. S .EQ. 0.00 ) RETURN
      DO 10 I = 1, M
          TEMP = C * X(I) + S * Y(INDX(I))
          Y(INDX(I)) = C * Y(INDX(I)) - S * X(I)
          X(I) = TEMP
   10 CONTINUE
      RETURN
      END
```

Example Apply a Givens rotation to x and y, where x is a sparse vector with interesting elements x_1, x_4, x_5, and x_9 stored in one-dimensional array X, and y is stored in a one-dimensional array Y of dimension 20.

```
INTEGER*4 M,INDX(4)
REAL*8    X(4),Y(20),C,S
DATA      INDX / 1, 4, 5, 9 /
M = 4
CALL DROTI (M,X,INDX,Y,C,S)
```

Name

SROTM/DROTM
Apply modified Givens rotation

Purpose

Given a modified Givens rotation matrix $H = \{h_{ij}\}$ as constructed by SROTMG or DROTMG, and real vectors x and y of length n, these subprograms apply the modified rotation

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad \text{for } i = 1, \ldots, n.$$

Refer to the description of the companion subprograms SROTMG and DROTMG for more details about the modified Givens rotation.

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays, and the indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          x(lenx), y(leny), param(5)
CALL SROTM(n, x, incx, y, incy, param)

INTEGER*4       n, incx, incy
REAL*8          x(lenx), y(leny), param(5)
CALL DROTM(n, x, incx, y, incy, param)
```

Input

n Number of elements of vectors x and y to be used. If **n** ≤ 0, the subprograms do not reference **x** or **y**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

- **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0: y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0: y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

param Array containing the matrix elements of the modified Givens rotation matrix H and a flag indicating which form the rotation matrix takes, and therefore which of the elements of **param** are significant. **param** is usually set by the companion subprogram SROTMG or DROTMG; refer to the description of these companion subprograms for the specific contents of **param**.

Output

x and y If **n** ≤ 0 or if **param**(1) = −2, **x** and **y** are unchanged. Otherwise, the result vectors overwrite the input.

Notes

The result is unspecified if **incx** = 0 or **incy** = 0 or if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

VECLIB also contains subprograms that construct and apply regular Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.

Example 1 Apply a modified Givens rotation to x and y, vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    X(20),Y(20),PARAM(5)
N = 10
INCX = 1
INCY = 1
CALL DROTM (N,X,INCX,Y,INCY,PARAM)
```

Example 2 Reduce 10-by-10 matrix **a** stored in two-dimensional array A of dimension 20-by-21 to upper-triangular form via modified Givens rotations (compare with "Example 2" on page 89 in the description of SROT and DROT).

```
      INTEGER*4 INCA,I,J,N
      REAL*8    A(20,21),D(20),PARAM(5)
      INCA = 20
      DO 10 I = 1, 10
         D(I) = 1.0D0
10    CONTINUE
      DO 30 I = 1, 9
         N = 10 - I
         DO 20 J = I+1, 10
             CALL DROTMG (D(I),D(J),A(I,I),A(J,I),PARAM)
             CALL DROTM  (N,A(I,I+1),INCA,A(J,I+1),INCA,PARAM)
20       CONTINUE
30    CONTINUE
      DO 40 I = 1, 10
         N = 11 - I
         CALL DSCAL (N,SQRT(D(I)),A(I,I),INCA)
40    CONTINUE
```

Name SROTMG/DROTMG
Construct modified Givens rotation

Purpose The Givens rotation, G, that annihilates z_1, if $z_1 \neq 0$ is

$$GW = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} w_1 & \cdots & w_n \\ z_1 & \cdots & z_n \end{bmatrix}$$

where $c = w_1/r$, $s = z_1/r$, and $r = \pm(w_1^2+z_1^2)^{1/2}$. Computing G and applying it to a pair of n vectors requires ~$4n$ floating-point multiplications, ~$2n$ floating-point additions, and one square root.

The modified Givens rotation is a device for reducing this operation count. Suppose that W above is available in factored form

$$W = D^{1/2}X = \begin{bmatrix} d_1^{1/2} & 0 \\ 0 & d_2^{1/2} \end{bmatrix} \cdot \begin{bmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{bmatrix}$$

These subprograms construct $\bar{d}_1$, $\bar{d}_2$, and H such that GW is obtained in the same factored form in which W was given

$$GW = \begin{bmatrix} \bar{d}_1^{1/2} & 0 \\ 0 & \bar{d}_2^{1/2} \end{bmatrix} \cdot \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{bmatrix}$$

H is chosen to have the same numerical stability as the standard Givens rotation but better computational efficiency. Thus, H usually has two elements equal to ±1. When this is true, computing H and applying it to a pair of n-vectors requires ~$2n$ floating-point multiplications, ~$2n$ floating-point additions, and no square roots. Companion VECLIB subprograms SROTM and DROTM are provided to apply the modified Givens notation to a pair of vectors.

In most applications, d_1 and d_2 are initially set to 1, manipulated by SROTMG or DROTMG as the modified Givens rotations are constructed, then applied to the vectors as the final step in the computation. For example, the reduction of an n-by-n matrix to upper-triangular form via modified Givens rotations requires $O(n)$ square roots compared to the $O(n^2)$ required by ordinary Givens rotations. Refer to "Example 2" in the description of SROTM and DROTM.

Usage

REAL*4 d1, d2, x1, y1, param(5)
CALL SROTMG(d1, d2, x1, y1, param)

REAL*8 d1, d2, x1, y1, param(5)
CALL DROTMG(d1, d2, x1, y1, param)

Input

d1 The scale factor for the "*x*" row.

d2 The scale factor for the "*y*" row.

x1 The first element of the "*x*" row.

y1 The first element of the "*y*" row. This is the element that is annihilated by the rotation.

Output

d1 The updated scale factor for the "*x*" row.

d2 The updated scale factor for the "*y*" row.

x1 The rotated first element of the "*x*" row.

param Array containing the matrix elements of the modified Givens rotation matrix *H* and a flag indicating which form the rotation matrix *H* takes and, therefore, which elements of **param** are significant. **param** is usually an argument to the companion subprogram SROTM or DROTM.

param(1) specifies the form of the rotation matrix *H*, as follows:

$$\mathbf{param}(1) = -2 \qquad H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{param}(1) = -1 \qquad H = \begin{bmatrix} \mathbf{param}(2) & \mathbf{param}(4) \\ \mathbf{param}(3) & \mathbf{param}(5) \end{bmatrix}$$

$$\mathbf{param}(1) = 0 \qquad H = \begin{bmatrix} 1 & \mathbf{param}(4) \\ \mathbf{param}(3) & 1 \end{bmatrix}$$

$$\mathbf{param}(1) = 1 \qquad H = \begin{bmatrix} \mathbf{param}(2) & 1 \\ -1 & \mathbf{param}(5) \end{bmatrix}$$

For each of the four values of **param**(1), only the indicated values of **param**(2) through **param**(5) are defined. The 0, 1, and –1 elements are not stored in **param**.

Notes VECLIB also contains subprograms that construct and apply ordinary Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.

Example Construct a modified Givens plane rotation that rotates vectors d_1x and d_2y in such a way as to annihilate d_2y_1. x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
REAL*8 D1,D2,X(20),Y(20),PARAM(5)
CALL DROTMG (D1,D2,X(1),Y(1),PARAM)
```

X(1) is the rotated result, so it should not be rotated by a subsequent call to DROTM.

Name SRSCL/DRSCL/CRSCL/CSRSCL/ZRSCL/ZDRSCL
Scale vector

Purpose Given a real or complex scalar a and a real or complex vector x of length n, these subprograms perform the reciprocal vector scaling operation

$x \leftarrow x \div a$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          a, x(lenx)
CALL SRSCL(n, a, x, incx)

INTEGER*4       n, incx
REAL*8          a, x(lenx)
CALL DRSCL(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*8       a, x(lenx)
CALL CRSCL(n, a, x, incx)

INTEGER*4       n, incx
REAL*4          a
COMPLEX*8       x(lenx)
CALL CSRSCL(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*16      a, x(lenx)
CALL ZRSCL(n, a, x, incx)

INTEGER*4       n, incx
REAL*8          a
COMPLEX*16      x(lenx)
CALL ZDRSCL(n, a, x, incx)
```

Input

n	Number of elements of vector x to be used in the scaling operation. If **n** ≤ 0, the subprograms do not reference **x**.
a	The scalar a, **a** ≠ 0.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

	incx	Increment for the array **x**, **incx** $\neq 0$. x is stored forward in array **x** with increment $\lvert$**incx**$\rvert$; that is, x_i is stored in **x**$((i-1)\times\lvert$**incx**$\rvert+1)$. Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.
Output	**x**	If **n** ≤ 0, then **x** is unchanged. Otherwise, $x \div a$ replaces the input.

Notes

The result is unspecified if **incx** = 0.

A divide-by-zero error occurs if **a** = 0 and **n** > 0.

Fortran Equivalent

```
      SUBROUTINE SRSCL (N,A, X,INCX)
      REAL*4 A,X(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
          X(IX) = X(IX) / A
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example

Scale the REAL*8 vector x by dividing by 2, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 2.0D0
CALL DRSCL (N,A,X,INCX)
```

Name SSCAL/DSCAL/CSCAL/CSSCAL/CSCALC/ZSCAL/ZDSCAL/ZSCALC
Scale vector

Purpose Given a real or complex scalar a and a real or complex vector x of length n, these subprograms perform the vector scaling operations

$x \leftarrow ax$ and $x \leftarrow a\bar{x}$

where $\bar{x}$ is the complex conjugate of x. The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          a, x(lenx)
CALL SSCAL(n, a, x, incx)

INTEGER*4       n, incx
REAL*8          a, x(lenx)
CALL DSCAL(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*8       a, x(lenx)
CALL CSCAL(n, a, x, incx)

INTEGER*4       n, incx
REAL*4          a
COMPLEX*8       x(lenx)
CALL CSSCAL(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*8       a, x(lenx)
CALL CSCALC(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*16      a, x(lenx)
CALL ZSCAL(n, a, x, incx)

INTEGER*4       n, incx
REAL*8          a
COMPLEX*16      x(lenx)
CALL ZDSCAL(n, a, x, incx)

INTEGER*4       n, incx
COMPLEX*16      a, x(lenx)
CALL ZSCALC(n, a, x, incx)
```

Input

n Number of elements of vector x to be used in the scaling operation. If $\mathbf{n} \leq 0$, the subprograms do not reference **x**.

a The scalar a.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x. x is used in conjugated form by CSCALC and ZSCALC and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array **x**, $\mathbf{incx} \neq 0$. x is stored forward in array **x** with increment $|\mathbf{incx}|$; that is, x_i is stored in $\mathbf{x}((i-1)\times|\mathbf{incx}|+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

x If $\mathbf{n} \leq 0$, then **x** is unchanged. Otherwise, ax replaces the input.

Notes The result is unspecified if **incx** = 0.

Fortran Equivalent

```
      SUBROUTINE SSCAL (N,A, X,INCX)
      REAL*4 A,X(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
          X(IX) = A * X(IX)
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Scale the REAL*8 vector x by 2, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 2.0D0
CALL DSCAL (N,A,X,INCX)
```

Name SSCTR/DSCTR/ISCTR/CSCTR/ZSCTR
Scatter sparse vector

Purpose Given a real, integer, or complex sparse vector x stored in compact form via a set of indices, these subprograms scatter those elements into the corresponding elements of a dense vector y stored in full storage form.

More precisely, let x be a sparse n-vector with $m \leq n$ *interesting* (usually nonzero) elements, and let $\{k_1, k_2, ..., k_m\}$ be the indices of these elements. If x is represented by arrays **x** and **indx** such that **indx**$(i) = k_i$ and **x**$(i) = x_{k_i}$, then

$y_{k_i} = \mathbf{x}_i, i = 1, 2, ..., m.$

Usage

```
INTEGER*4       m, indx(m)
REAL*4          x(m), y(n)
CALL SSCTR(m, x, indx, y)

INTEGER*4       m, indx(m)
REAL*8          x(m), y(n)
CALL DSCTR(m, x, indx, y)

INTEGER*4       m, indx(m), x(m), y(n)
CALL ISCTR(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*8       x(m), y(n)
CALL CSCTR(m, x, indx, y)

INTEGER*4       m, indx(m)
COMPLEX*16      x(m), y(n)
CALL ZSCTR(m, x, indx, y)
```

Input

m	Number of interesting elements, **m** $\leq$ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.
x	Array of length **m** containing the interesting elements of x. **x**$(j) = x_i$ if **indx**$(j) = i$.

	indx	Array containing the indices $\{k_i\}$ of the interesting elements of x. The indices must satisfy the following: $1 \le \mathbf{indx}(i) \le \mathbf{n},\ i = 1, 2, \ldots, \mathbf{m}$ $\mathbf{indx}(i) \ne \mathbf{indx}(j),\ 1 \le i \ne j \le \mathbf{m}$ where **n** is the length of **y**.
Output	**y**	Array containing the elements of y, $\mathbf{y}(i) = y_i$. If $\mathbf{m} \le 0$, then **y** is unchanged. Otherwise, only the elements of **y** whose indices are included in **indx** are changed.

Notes The result is unspecified if any element of **indx** is out of range, if any two elements of **indx** have the same value, or if **x**, **indx**, and **y** overlap such that any element of x or any index shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SSCTR (M, X,INDX, Y)
      REAL*4 X(*),Y(*)
      INTEGER*4 INDX(*)
      IF ( M .LE. 0 ) RETURN
      DO 10 I = 1, M
          Y(INDX(I)) = X(I)
   10 CONTINUE
      RETURN
      END
```

Example Scatter x into y, where x is a sparse vector with interesting elements x_1, x_4, x_5, and x_9 stored in one-dimensional array X, and y is stored in a one-dimensional array Y of dimension 20.

```
INTEGER*4 M,INDX(4)
REAL*8    X(4),Y(20)
DATA      INDX / 1, 4, 5, 9 /
M = 4
CALL DSCTR (M,X,INDX,Y)
```

Name

SSUM/DSUM/ISUM/CSUM/ZSUM
Vector sum

Purpose

Given a real, integer, or complex vector x of length n, these subprograms compute the sum of the elements of the vector

$$s = \sum_{i=1}^{n} x_i.$$

The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          s, SSUM, x(lenx)
s = SSUM(n, x, incx)

INTEGER*4       n, incx
REAL*8          s, DSUM, x(lenx)
s = DSUM(n, x, incx)

INTEGER*4       n, incx, s, ISUM, x(lenx)
s = ISUM(n, x, incx)

INTEGER*4       n, incx
COMPLEX*8       s, CSUM, x(lenx)
s = CSUM(n, x, incx)

INTEGER*4       n, incx
COMPLEX*16      s, ZSUM, x(lenx)
s = ZSUM(n, x, incx)
```

Input

n	Number of elements of vector x to be used in the sum. If **n** ≤ 0, the subprograms do not reference **x**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.
incx	Increment for the array **x**. x is stored forward in array **x** with increment |**incx**|; that is, x_i is stored in **x**((*i*−1)×|**incx**|+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s	If $\mathbf{n} \leq 0$, then $\mathbf{s} = 0$. Otherwise, **s** is the sum of the elements of x.

Fortran Equivalent

```
      REAL*4 FUNCTION SSUM (N, X,INCX)
      REAL*4 X(*)
      SSUM = 0.0
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
          SSUM = SSUM + X(IX)
          IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Compute the sum of the elements of a REAL*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*4 N,INCX
REAL*8    S,X(20)
N = 10
INCX = 1
S = DSUM (N,X,INCX)
```

Name SSWAP/DSWAP/ISWAP/CSWAP/ZSWAP
Swap two vectors

Purpose Given real, integer, or complex vectors x and y of length n, these subprograms perform the vector interchange operation

$x \leftrightarrow y$.

The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. The indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incx, incy
REAL*4          x(lenx), y(leny)
CALL SSWAP(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
REAL*8          x(lenx), y(leny)
CALL DSWAP(n, x, incx, y, incy)

INTEGER*4       n, incx, incy, x(lenx), y(leny)
CALL ISWAP(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*8       x(lenx), y(leny)
CALL CSWAP(n, x, incx, y, incy)

INTEGER*4       n, incx, incy
COMPLEX*16      x(lenx), y(leny)
CALL ZSWAP(n, x, incx, y, incy)
```

Input

n	Number of elements of vectors x and y to be used in the swap operation. If **n** ≤ 0, the subprograms do not reference **x** or **y**.
x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

- **incx** > 0: x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0: x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0: y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0: y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

x and **y** If **n** ≤ 0, then **x** and **y** are unchanged. Otherwise, x and y are interchanged in **x** and **y**.

Notes

The result is unspecified if **incx** = 0 or **incy** = 0 or if **x** and **y** overlap such that any element of x shares a memory location with any element of y.

Fortran Equivalent

```
      SUBROUTINE SSWAP (N, X,INCX, Y,INCY)
      REAL*4 TEMP,X(*),Y(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IY = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          TEMP = X(IX)
          X(IX) = Y(IY)
          Y(IY) = TEMP
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Interchange REAL*8 vectors x and y, where x and y are vectors 10 elements long stored in one-dimensional arrays X and Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*8    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL DSWAP (N,X,INCX,Y,INCY)
```

Example 2 Interchange rows 3 and 6 of a 10-by-10 matrix **a** stored in two-dimensional array A of dimension 20-by-21.

```
INTEGER*4 N,INCA
REAL*8    A(20,21)
N = 10
INCA = 20
CALL DSWAP (N,A(3,1),INCA,A(6,1),INCA)
```

Name

SWDOT/DWDOT/CWDOTC/CWDOTU/ZWDOTC/ZWDOTU
Weighted dot product

Purpose

Given a real weight vector w and real or complex data vectors x and y, all of length n, these subprograms compute the weighted dot products

$$s = \sum_{i=1}^{n} w_i x_i y_i \quad \text{and} \quad s = \sum_{i=1}^{n} w_i \bar{x}_i y_i$$

where $\bar{x}$ is the complex conjugate of x. The vectors can be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays, and the indexing through the arrays can be either forward or backward.

Usage

```
INTEGER*4       n, incw, incx, incy
REAL*4          s, SWDOT, w(lenw), x(lenx), y(leny)
s = SWDOT(n, w, incw, x, incx, y, incy)

INTEGER*4       n, incw, incx, incy
REAL*8          s, DWDOT, w(lenw), x(lenx), y(leny)
s = DWDOT(n, w, incw, x, incx, y, incy)

INTEGER*4       n, incw, incx, incy
REAL*4          w(lenw)
COMPLEX*8       s, CWDOTC, x(lenx), y(leny)
s = CWDOTC(n, w, incw, x, incx, y, incy)

INTEGER*4       n, incw, incx, incy
REAL*4          w(lenw)
COMPLEX*8       s, CWDOTU, x(lenx), y(leny)
s = CWDOTU(n, w, incw, x, incx, y, incy)

INTEGER*4       n, incw, incx, incy
REAL*8          w(lenw)
COMPLEX*16      s, ZWDOTC, x(lenx), y(leny)
s = ZWDOTC(n, w, incw, x, incx, y, incy)

INTEGER*4       n, incw, incx, incy
REAL*8          w(lenw)
COMPLEX*16      s, ZWDOTU, x(lenx), y(leny)
s = ZWDOTU(n, w, incw, x, incx, y, incy)
```

Input

n Number of elements of vectors w, x, and y to be used in the dot product. If **n** ≤ 0, the subprograms do not reference **w**, **x**, or **y**.

w Array of length **lenw** = (**n**−1)×|**incw**|+1 containing the n-vector w.

incw Increment for the array **w**:

incw ≥ 0 w is stored forward in array **w**; that is, w_i is stored in **w**((*i*−1)×**incw**+1).

incw < 0 w is stored backward in array **w**; that is, w_i is stored in **w**((*i*−**n**)×**incw**+1).

Use **incw** = 1 if the vector w is stored contiguously in array **w**; that is, if w_i is stored in **w**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CWDOTC and ZWDOTC and in unconjugated form by the other subprograms.

incx Increment for the array **x**:

incx ≥ 0 x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).

incx < 0 x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**:

incy ≥ 0 y is stored forward in array **y**; that is, y_i is stored in **y**$((i-1)\times$**incy**$+1)$.

incy < 0 y is stored backward in array **y**; that is, y_i is stored in **y**$((i-$**n**$)\times$**incy**$+1)$.

Use **incy** = 1 if the vector y is stored contiguously in array **y**; that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

s The resulting value of the weighted dot product. If **n** ≤ 0, then **s** = 0. Otherwise,

$$\mathbf{s} = \sum_{i=1}^{n} w_i x_i y_i$$

unless the subprogram name is CWDOTC or ZWDOTC, in which case

$$\mathbf{s} = \sum_{i=1}^{n} w_i \bar{x}_i y_i.$$

Notes If **incw** = 0, then w_i = **w**(1) for all i. If **incx** = 0, then x_i = **x**(1) for all i. If **incy** = 0, then y_i = **y**(1) for all i. In any of these cases, another VECLIB dot product subprogram would be more efficient.

Fortran Equivalent

```
      REAL*4 FUNCTION SWDOT (N, W,INCW, X,INCX, Y,INCY)
      REAL*4 W(*),X(*),Y(*)
      SWDOT = 0.0
      IF ( N .LE. 0 ) RETURN
      IW = 1
      IX = 1
      IY = 1
      IF ( INCW .LT. 0 ) IW = 1 - (N-1) * INCW
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
      DO 10 I = 1, N
          SWDOT = SWDOT + W(IW) * X(IX) * Y(IY)
          IW = IW + INCW
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example 1 Compute the REAL*8 weighted dot product

$$s = \sum_{i=1}^{10} w_i x_i y_i,$$

where w, x, and y are vectors 10 elements long stored in one-dimensional arrays W, X, and Y, of dimension 20.

```
INTEGER*4 N,INCW,INCX,INCY
REAL*8    S,DWDOT,W(20),X(20),Y(20)
N = 10
INCW = 1
INCX = 1
INCY = 1
S = DWDOT (N,W,INCW,X,INCX,Y,INCY)
```

Example 2 Compute the REAL*8 weighted dot product

$$s = \sum_{i=1}^{10} w_i x_i y_i,$$

where w and y are vectors 10 elements long stored in one-dimensional arrays W and Y of dimension 20, and x is the 4th row of a 10-by-10 matrix stored in a two-dimensional array X of dimension 20-by-21.

```
INTEGER*4 N
REAL*8    S,DWDOT,W(20),X(20,21),Y(20)
N = 10
S = DWDOT (N,W,1,X(I,1),20,Y,1)
```

Name SZERO/DZERO/IZERO/CZERO/ZZERO
Clear vector

Purpose These subprograms set all of the elements of a real, integer, or complex n-vector x to zero. The vector can be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
INTEGER*4       n, incx
REAL*4          x(lenx)
CALL SZERO(n, x, incx)

INTEGER*4       n, incx
REAL*8          x(lenx)
CALL DZERO(n, x, incx)

INTEGER*4       n, incx, x(lenx)
CALL IZERO(n, x, incx)

INTEGER*4       n, incx
COMPLEX*8       x(lenx)
CALL CZERO(n, x, incx)

INTEGER*4       n, incx
COMPLEX*16      x(lenx)
CALL ZZERO(n, x, incx)
```

Input

n Number of elements of vector x to be set to zero. If **n** ≤ 0, the subprograms do not reference **x**.

incx Increment for the array **x**, **incx** $\neq 0$. x is stored forward in array **x** with increment $|$**incx**$|$; that is, x_i is stored in **x**$((i-1)\times|$**incx**$|+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**; that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

Output

x Array of length **lenx** = (**n**−1)×$|$**incx**$|+1$ containing the n-vector x that has been set to zero. If **n** ≤ 0, then **x** is unchanged. Otherwise, $x \leftarrow 0$.

Fortran Equivalent

```
      SUBROUTINE SZERO (N, X,INCX)
      REAL*4 X(*)
      IF ( N .LE. 0 ) RETURN
      IX = 1
      INCXA = ABS ( INCX )
      DO 10 I = 1, N
         X(IX) = 0.0
         IX = IX + INCXA
   10 CONTINUE
      RETURN
      END
```

Example Zero the REAL*8 vector, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20 (compare with "Example 2" in the description of SCOPY).

```
INTEGER*4 N,INCX
REAL*8    X(20)
N = 10
INCX = 1
CALL DZERO (N,X,INCX)
```

BLAS Standard routines

Name F_SAMAX_VAL/F_DAMAX_VAL/F_CAMAX_VAL/F_ZAMAX_VAL
Maximum absolute value and location

Purpose F_xAMAX_VAL returns the largest component of the vector x with respect to the absolute value, and also returns the offset or index of the largest component of the vector x. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index (zero) and r to zero. The resulting scalar r is always real.

$$k, x_k \text{ such that } k = \arg_{0 \le i < n} \max(|Re(x_i)| + |Im(x_i)|)$$

Usage

```
INTEGER         INCX, K, N
REAL*4          R, X( * )
SUBROUTINE F_SAMAX_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*8          R, X( * )
SUBROUTINE F_DAMAX_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMAX_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*8          R
COMPLEX*16      X( * )
SUBROUTINE F_ZAMAX_VAL (N, X, INCX, K, R)
```

Input

N	Number of elements of vector x.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output **K** Displacement returned by the routine. The smallest offset or index such that

$x_k = \max(|Re(x_i)| + |Im(x_i)|)$ (where $0 \le i < n$)

R REAL scalar. The largest component of the vector x.

Name

F_SAMIN_VAL/F_DAMIN_VAL/F_CAMIN_VAL/F_ZAMIN_VAL
Minimum absolute value and location

Purpose

F_xAMIN_VAL returns the smallest component of the vector x with respect to the absolute value and also returns the offset or index of the smallest component of the vector x. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index (zero), and r to zero. The resulting scalar r is always real.

$$k, x_k \text{ such that } k = \arg_{0 \le i < n} \min(|Re(x_i)| + |Im(x_i)|)$$

Usage

```
INTEGER         INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SAMIN_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DAMIN_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMIN_VAL (N, X, INCX, K)

INTEGER         INCX, K, N
REAL*8          R
COMPLEX*16      X( * )
SUBROUTINE F_ZAMIN_VAL (N, X, INCX, K)
```

Input

N Number of elements of vector x.

X REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component x_i, $i = 1,\ldots, n$, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

K Displacement returned by the routine. The smallest offset or index such that

$$x_k = \min(|Re(x_i)| + |Im(x_i)|) \qquad (\text{where } 0 \le i < n)$$

R REAL scalar. The smallest component of the vector x.

Name

F_SAPPLY_GROT/F_DAPPLY_GROT/F_CAPPLY_GROT/F_ZAPPLY_GROT
Apply plane rotation

Purpose

F_xAPPLY_GROT applies a plane rotation to the vectors x and y. When the vectors x and y are real vectors, the scalars c and s are real scalars. When the vectors x and y are complex vectors, c is a real scalar and s is a complex scalar.

$$\forall i \in [0 \ldots n-1], \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \bullet \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

If n is less then or equal to zero or if c is one and s is zero, F_xAPPLY_GROT returns immediately.

Usage

```
INTEGER         INCX, INCY, N
REAL*4          C, S, X( * ), Y( * )
SUBROUTINE F_SAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER         INCX, INCY, N
REAL*8          C, S, X( * ), Y( * )
SUBROUTINE F_DAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER         INCX, INCY, N
REAL*4          C
COMPLEX*8       S, X( * ), Y( * )
SUBROUTINE F_CAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER         INCX, INCY, N
REAL*8          C
COMPLEX*16      S, X( * ), Y( * )
SUBROUTINE F_ZAPPLY_GROT (N, C, S, X, INCX, Y, INCY)
```

Input	**N**	Number of elements of vector x.
	C	REAL scalar.
	S	REAL or COMPLEX scalar.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**X**	The updated array replaces the input—The plane rotation of input X.
	Y	The updated array replaces the input—The plane rotation of input Y.

Name F_SAXPBY/F_DAXPBY/F_CAXPBY/F_ZAXPBY
Scaled vector accumulation

Purpose F_xAXPBY scales the vector x by α and the vector y by β, adds these two vectors, and stores the result in the vector y. If n is less than or equal to zero, or if α is equal to zero and β is equal to one, the routine returns immediately.

$$y \leftarrow \alpha x + \beta y$$

Usage

```
INTEGER        INCX, INCY, N
REAL*4         ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_SAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER        INCX, INCY, N
REAL*8         ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_DAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER        INCX, INCY, N
COMPLEX*8      ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_CAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER        INCX, INCY, N
COMPLEX*16     ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_ZAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)
```

Input

N	Number of elements of vector x.
ALPHA	The scalar ALPHA.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
BETA	The scalar BETA.
Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.

	INCY	Increment for the array y. A vector y having component y_i, $i = 1,..., n$, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated array replaces the input.

$$y \leftarrow \alpha x + \beta y$$

Name

F_SAXPY_DOT/F_DAXPY_DOT/F_CAXPY_DOT/F_ZAXPY_DOT
Combine AXPY and DOT routines

Purpose

F_xAXPY_DOT combines an AXPY and a DOT product. This routine first decrements w by a multiple of v, and then computes a dot product using w. If n is less than or equal to zero, the routine returns immediately.

$$\hat{w} \leftarrow w - \alpha v$$

$$r \leftarrow \hat{w}^T u$$

Combining two BLAS-1 calls reduces overhead and transfer of data in modified Gram-Schmidt orthogonalization.

Usage

```
INTEGER          INCW, INCV, INCU, N
REAL*4           ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_SAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER          INCW, INCV, INCU, N
REAL*8           ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_DAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER          INCW, INCV, INCU, N
COMPLEX*8        ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_CAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER          INCW, INCV, INCU, N
COMPLEX*16       ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_ZAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )
```

Input

N	Number of elements of vector.
ALPHA	The scalar ALPHA.
W	REAL or COMPLEX array, minimum length (N - 1) x |**incw**| + 1.
INCW	Increment for the array w. A vector w having component w_i, i = 1,...,n, is stored in an array W() with increment argument **incw**. If **incw** > 0 then w_i is stored in W (1 + (i - 1) x **incw**). If **incw** < 0 then w_i is stored in W(1 + (N - i) x |**incw**|). **incw** = 0 is an illegal value.

	V	REAL or COMPLEX array, minimum length (**N** - 1) x |**incv**| + 1.
	INCV	Increment for the array *v*. A vector v having component v_i, $i = 1,..., n$, is stored in an array **V**() with increment argument **incv**. If **incv** > 0 then v_i is stored in **V**(1 + (*i* - 1) x **incv**). If **incv** < 0 then v_i is stored in **V**(1 + (**N** - i) x |**incv**|). **incv** = 0 is an illegal value.
	U	REAL or COMPLEX array, minimum length (**N** - 1) x |**incu**| + 1.
	INCU	Increment for the array *u*. A Vector *u* having component u_i, $i = 1,..., n$, is stored in an array U() with increment argument **incu**. If **incu** > 0 then u_i is stored in **U**(1 + (*i* - 1) x **incu**). If **incu** < 0 then u_i is stored in **U**(1 + (**N** - i) x |**incu**|). **incu** = 0 is an illegal value.
Output	**R**	REAL or COMPLEX result of the operation.

Name F_SCOPY/F_DCOPY/F_CCOPY/F_ZCOPY
Copy vector

Purpose F_xCOPY copies the vector x into the vector y, that is,

$$y \leftarrow x$$

If n is less than or equal to zero, the routine returns immediately.

Usage

```
INTEGER          INCX, INCY, N
REAL*4           X( * ), Y( * )
SUBROUTINE F_SCOPY (N, X, INCX, Y, INCY)

INTEGER          INCX, INCY, N
REAL*8           X( * ), Y( * )
SUBROUTINE F_DCOPY (N, X, INCX, Y, INCY)

INTEGER          INCX, INCY, N
COMPLEX*8        X( * ), Y( * )
SUBROUTINE F_CCOPY (N, X, INCX, Y, INCY)

INTEGER          INCX, INCY, N
COMPLEX*16       X( * ), Y( * )
SUBROUTINE F_ZCOPY (N, X, INCX, Y, INCY)
```

Input

N	Number of elements of vector x.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.

Output

Y	The result of the copy of vector x to vector y. REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.

Name F_SDOT/F_DDOT/F_CDOT/F_ZDOT
Add scaled dot product

Purpose F_xDOT adds the scaled dot product of two vectors x and y into a scaled scalar r. The routine returns immediately if n is less than zero, or, if β is equal to one and either α or n is equal to zero. If α is equal to zero then x and y are not read. Similarly, if β is equal to zero, r is not referenced.

When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument *conj*. If x and y are real vectors, the operator argument *conj* has no effect.

$$r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i$$

$$r \leftarrow \beta r + \alpha x^H y = \beta r + \alpha \sum_{i=0}^{n-1} \overline{x_i} y_i$$

Usage

```
INTEGER         CONJ, INCX, INCY, N
REAL*4          ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_SDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER         CONJ, INCX, INCY, N
REAL*8          ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_DDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER         CONJ, INCX, INCY, N
COMPLEX*8       ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_CDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER         CONJ, INCX, INCY, N
COMPLEX*16      ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_ZDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)
```

Input	**CONJ**	Specifies conjugation for vector components in complex routines. Vector components are used conjugated or unconjugated. Use either **BLAS_CONJ** or **BLAS_NO_CONJ**. When x and y are real vectors the **conj** operator argument has no effect.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**R**	REAL or COMPLEX scalar. The scaled dot product of the two input vectors x and y.

Name F_SFPINFO/F_DFPINFO
Environmental inquiry

Purpose F_xFPINFO queries for machine-specific floating point characteristics. For BLAS Standard routines, error bounds and limitations due to overflow and underflow depend on details of how floating point numbers are represented. These details are available by calling F_xFPINFO.

Usage

```
INTEGER      CMACH
REAL*4       FUNCTION F_SFPINFO (CMACH)

INTEGER      CMACH
REAL*4       FUNCTION F_DFPINFO (CMACH)
```

Input

CMACH

A named integer constant. The names for the **CMACH** argument are given in the appropriate language's include file.

Table 2-1 describes the **CMACH** floating point parameters, the corresponding BLAS Standard named constant (from the Fortran 77 include file), and the values returned by F_xFPINFO.

Table 2-1 describes floating point parameters and values returned by FPINFO.

Table 2-1 FPINFO return values

Floating point parameter	Fortran77 named constant	Description	Value in IEEE single	Value in IEEE double
BASE	BLAS_BASE	Base of machine	2	2
T	BLAS_T	Number of digits	24	53
RND	BLAS_RND	Equals 1 when proper rounding occurs in addition. Otherwise, equals 0	1	1
IEEE	BLAS_IEEE	Equals 1 when rounding in addition is IEEE style. Otherwise, equals 0	1	1
EMIN	BLAS_EMIN	Minimum exponent before (gradual) underflow	-126	-1022
EMAX	BLAS_EMAX	Minimum exponent before overflow	127	1023
EPS	BLAS_EPS	Machine epsilon $EPS = 0.5BASE^{1-T}$ if RND = 1 $EPS = BASE^{1-T}$ if RND = 0	$2^{-24} \approx 5e-8$	$2^{-53} \approx 1e-16$
PREC	BLAS_PREC	EPS*BASE	2^{-23}	2^{-52}
UN	BLAS_UNDERFLOW	Underflow threshold $= BASE^{EMIN}$	$2^{-126} \approx 1e-38$	$2^{-1022} \approx 1e-308$
OV	BLAS_OVERFLOW	Overflow threshold $= BASE^{EMAX+1} \times (1-EPS)$	$2^{128} \approx 1e38$	$2^{1024} \approx 1e308$
SFMIN	BLAS_SFMIN	Safe minimum, such that 1/SFMIN does not overflow. If 1/OV < UN, SFMIN = UN. Otherwise, SFMIN = (1+EPS) / OV	$2^{-126} \approx 1e-38$	$2^{-1022} \approx 1e-308$

Name F_SGEN_GROT/F_DGEN_GROT/F_CGEN_GROT/F_ZGEN_GROT
Generate Givens rotation

Purpose F_xGEN_GROT constructs a Givens plane rotation so that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \bullet \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where c is always a real scalar and

$$c^2 + |s|^2 = 1$$

The scalars a and b are unchanged on exit. If b is equal to zero, then the pair (c,s) is chosen to be equal to (1, 0). Otherwise, when a and b are real scalars and a is equal to zero, the pair (c,s) is chosen to be equal to (0, 1); when a and b are complex scalars and a is equal to zero, c is chosen to be zero and s is chosen so that r is real.

$$(c, s, r) \leftarrow rot(a, b)$$

Usage

REAL*4 A, B, R, S, C
SUBROUTINE F_SGEN_GROT (A, B, C, S, R)

REAL*8 A, B, R, S, C
SUBROUTINE F_DGEN_GROT (A, B, C, S, R)

COMPLEX*8 A, B, R, S
SUBROUTINE F_CGEN_GROT (A, B, C, S, R)

COMPLEX*16 A, B, R, S
SUBROUTINE F_ZGEN_GROT (A, B, C, S, R)

Input

A	REAL or COMPLEX scalar. A is unchanged on exit.
B	REAL or COMPLEX scalar. B is unchanged on exit.

Output

C	REAL scalar.
R	The REAL or COMPLEX product of the operation.
S	REAL or COMPLEX scalar.

Name F_SGEN_HOUSE/F_DGEN_HOUSE/F_CGEN_HOUSE/F_ZGEN_HOUSE
Generate Householder transform

Purpose F_xGEN_HOUSE generates an elementary reflector H of order n, s such that

$$H\begin{bmatrix}\alpha\\ x\end{bmatrix} = \begin{bmatrix}\beta\\ 0\end{bmatrix} \text{ and } H^*H = I$$

where α and β are scalars, and x is an $(n - 1)$-element vector. β is always a real scalar. H is represented in the following form:

$$H = I - \tau\begin{bmatrix}1\\ \upsilon\end{bmatrix}(1 \quad \upsilon^T)$$

where τ is a scalar and υ is an $(n - 1)$-element vector. τ is called the Householder scalar and $\begin{bmatrix}1\\ \upsilon\end{bmatrix}$ the Householder vector.

Note that when x is a complex vector, H is not Hermitian. If the elements of x are zero, and α is real, then τ is equal to zero and H is the unit matrix. Otherwise, the real part of τ is greater than or equal to one, and less than or equal to two. Moreover, the absolute value of the quantity $(\tau - 1)$ is less than or equal to one.

On exit, the scalar argument **alpha** is overwritten with the value of the scalar β. Similarly, the vector argument **x** is overwritten with the vector υ. If n is less than or equal to zero, this function returns immediately with the output scalar **tau** set to zero.

Usage

```
INTEGER      INCX, N
REAL*4       ALPHA, TAU
REAL*4       X ( * )
SUBROUTINE F_SGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER      INCX, N
REAL*8       ALPHA, TAU
REAL*8       X ( * )
SUBROUTINE F_DGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER      INCX, N
COMPLEX*8    ALPHA, TAU
COMPLEX*8    X ( * )
SUBROUTINE F_CGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER      INCX, N
COMPLEX*16   ALPHA, TAU
COMPLEX*8    X ( * )
SUBROUTINE F_ZGEN_HOUSE ( N, ALPHA, X, INCX, TAU )
```

Input

N	Number of elements of vector x.
ALPHA	REAL or COMPLEX scalar.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1).
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

TAU	REAL or COMPLEX scalar. If n = 0, F_xGEN_HOUSE returns immediately with **TAU** set to zero.

Name F_SGEN_JROT/F_DGEN_JROT/F_CGEN_JROT/F_ZGEN_JROT
Generate Jacobi rotation

Purpose F_xGEN_JROT constructs a Jacobi rotation so that $(a, b, c, s) \leftarrow jrot(x, y, z)$

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} = \begin{bmatrix} c & \bar{s} \\ -s & c \end{bmatrix} \bullet \begin{bmatrix} x & y \\ \bar{y} & z \end{bmatrix} \bullet \begin{bmatrix} c & -\bar{s} \\ s & c \end{bmatrix}$$

Input the **JROT** parameter to specify whether the rotation generated is outer, inner, or sorted.

- If **JROT = BLAS_INNER_ROTATION**

  then the rotation is chosen so that $c \geq \frac{1}{\sqrt{2}}$

- If **JROT = BLAS_OUTER_ROTATION**

  then the rotation is chosen so that $0 \leq c \leq \frac{1}{\sqrt{2}}$

- `If` **JROT = BLAS_SORTED_ROTATION**

  then the rotation is chosen so that $abs(a) \geq abs(b)$

Usage

```
INTEGER         JROT
REAL*4          S, X, Y, Z, C
SUBROUTINE F_SGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER         JROT
REAL*8          S, X, Y, Z, C
SUBROUTINE F_DGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER         JROT
REAL*4          X, Z
COMPLEX*8       S, Y
SUBROUTINE F_CGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER         JROT
REAL*8          X, Z
COMPLEX*16      S, Y
SUBROUTINE F_ZGEN_JROT (JROT, X, Y, Z, C, S)
```

Input	**JROT**	Specifies whether the Jacobi rotation generated is OUTER, INNER, or SORTED. The following are valid options: • **BLAS_INNER_ROTATION** • **BLAS_OUTER_ROTATION** • **BLAS_SORTED_ROTATION**
	X	REAL scalar. X is replaced by A on exit.
	Y	REAL or COMPLEX scalar. Y is unchanged on exit.
	Z	REAL scalar. Z is replaced by B on exit.
Output	**A**	REAL scalar. Replaces the input X.
	B	REAL scalar. Replaces the input Z.
	C	REAL cosine used to apply the Jacobi rotation.
	S	REAL or COMPLEX sine used to apply the Jacobi rotation.

Name F_SMAX_VAL/F_DMAX_VAL
Maximum value and location

Purpose F_xMAX_VAL returns the largest component of a real vector x and also the smallest offset or index k.

$$k, x_k \text{ such that } k = \arg_{0 \le i < n} \max(x_i)$$

When the value of the n argument is less than or equal to zero, F_xMAX_VAL initializes the output scalars k to the largest invalid index (zero) and r to zero.

The routine F_xMIN_VAL operates strictly on real vectors and is not defined for complex vectors.

Usage

```
INTEGER        INCX, K, N
REAL*4         R
REAL*4         X( * )
SUBROUTINE F_SMAX_VAL (N, X, INCX, K, R)

INTEGER        INCX, K, N
REAL*8         R
REAL*8         X( * )
SUBROUTINE F_DMAX_VAL (N, X, INCX, K, R)
```

Input

N Number of elements of vector x.

X REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

K Displacement returned by the routine. The smallest offset or index such that $x_k = max_{0 \le i < n} x_i$

R REAL scalar. The largest component of the REAL vector x.

Name F_SMIN_VAL/F_DMIN_VAL
Minimum value and location

Purpose F_xMIN_VAL returns the smallest component of a real vector x and also the smallest offset or index k.

$$k, x_k \text{ such that } k = \arg_{0 \le i < n} \max(x_i)$$

When the value of the n argument is less than or equal to zero, F_xMIN_VAL initializes the output scalars k to the largest invalid index (zero) and r to zero. When the value of the n argument is less than or equal to zero, F_xMIN_VAL initializes the output scalars k to the largest invalid index (zero) and r to zero.

The routine F_xMIN_VAL operates strictly on real vectors and is not defined for complex vectors.

Usage

```
INTEGER         INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SMIN_VAL (N, X, INCX, K, R)

INTEGER         INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DMIN_VAL (N, X, INCX, K, R)
```

Input

N	Number of elements of vector x.
X	REAL array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

K	Displacement returned by the routine. The smallest offset or index such that $x_k = min_{0 \le i < n} x_i$
R	REAL scalar. The smallest component of the REAL vector x.

Name

F_SNORM/F_DNORM
Norm of a vector

Purpose

F_xNORM computes one of the following for a vector x depending on the value passed as the norm operator argument:

- 1-norm
- Real 1-norm
- 2-norm
- Frobenius-norm
- Max-norm
- Real-max-norm
- Infinity-norm
- Real infinity-norm

$$r \leftarrow \|x\|_1, \|x\|_{1R}, \|x\|_2, \|x\|_\infty, or \|x\|_{\infty R}$$

Refer to **NORM** parameter below for details. If n is less than or equal to zero, this routine returns immediately with the output scalar r set to zero. The resulting scalar r is always real.

Usage

```
INTEGER     INCX, N, NORM
REAL*4      X( * )
REAL*4      FUNCTION F_SNORM (NORM, N, X, INCX)

INTEGER     INCX, N, NORM
REAL*8      X( * )
REAL*4      FUNCTION F_SNORM (NORM, N, X, INCX)
```

Input

NORM Specifies the norm to compute. Seven distinct values are possible, namely the 1-norm, real 1-norm, infinity-norm, and real infinity-norm for vectors and matrices, the 2-norm for vectors, and the Frobenius-norm, max-norm, and real max-norm for matrices. Use one of the following constants:

BLAS_ONE_NORM	**BLAS_REAL_ONE_NORM**
BLAS_INF_NORM	**BLAS_REAL_INF_NORM**
BLAS_TWO_NORM	**BLAS_FROBENIUS_NORM**
BLAS_MAX_NORM	**BLAS_REAL_MAX_NORM**

When you specify **NORM = BLAS_FROBENIUS_NORM**, an error flag is not raised, and the routine returns the two-norm.

N Number of elements of vector x.

X REAL array, minimum length (N-1) x |**incx**|+ 1.

INCX Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Name

F_SPERMUTE/F_DPERMUTE/F_CPERMUTE/F_ZPERMUTE
Permute vector

Purpose

F_xPERMUTE permutes the entries of a vector x according to the permutation vector p. If n is less than or equal to zero, the routine returns immediately.

An n-by-n permutation matrix P is represented as a product of at most n interchange permutations. An interchange permutation E is a permutation obtained by swapping two rows of the identity matrix. An efficient way to represent a general permutation matrix P is with an integer vector p of length n. In other words, $P = E_n...E_1$ and each E_i is the identity with rows i and p_i interchanged:

For $i = n$ to 1 and $incp < 0$, $x(i) \leftrightarrow x(p(i))$

For $i = 1$ to n and $incp > 0$, $x(i) \leftrightarrow x(p(i))$

Usage

```
INTEGER        INCP, INCX, N
INTEGER        P( * )
REAL*4         X( * )
SUBROUTINE F_SPERMUTE (N, P, INCP, X, INCX)

INTEGER        INCP, INCX, N
INTEGER        P( * )
REAL*8         X( * )
SUBROUTINE F_DPERMUTE (N, P, INCP, X, INCX)

INTEGER        INCP, INCX, N
INTEGER        P( * )
COMPLEX*8      X( * )
SUBROUTINE F_CPERMUTE (N, P, INCP, X, INCX)

INTEGER        INCP, INCX, N
INTEGER        P( * )
COMPLEX*16     X( * )
SUBROUTINE F_ZPERMUTE (N, P, INCP, X, INCX)
```

Input	**N**	Number of elements of vector x.
	P	REAL array, minimum length (**N** - 1) x |**incp**| + 1.
	INCP	Increment for the array p. A vector p having component p_i, i = 1,..., n, is stored in an array **P**() with increment argument **incp**. The value of **incp** can be positive or negative. A negative value of **incp** applies the permutation in the opposite direction. **incp** = 0 is an illegal value.
	X	REAL array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Name F_SRSCALE/F_DRSCALE/F_CRSCALE/F_ZRSCALE
Reciprocal Scale

Purpose F_xRSCALE scales the entries of a vector x by the real scalar $1/\alpha$. The scalar α is always real and should be nonzero. Scaling is done without overflow or underflow as long as the result, x/α, does not overflow or underflow. If n is less than or equal to zero, this routine returns immediately.

$$x \leftarrow x/\alpha$$

Usage

```
INTEGER        INCX, N
REAL*4         ALPHA, X( * )
SUBROUTINE F_SRSCALE (N, ALPHA, X, INCX)

INTEGER        INCX, N
REAL*8         ALPHA, X( * )
SUBROUTINE F_DRSCALE (N, ALPHA, X, INCX)

INTEGER        INCX, N
REAL*4         ALPHA
COMPLEX*8      X( * )
SUBROUTINE F_CRSCALE (N, ALPHA, X, INCX)

INTEGER        INCX, N
REAL*4         ALPHA
COMPLEX*8      X( * )
SUBROUTINE F_ZRSCALE (N, ALPHA, X, INCX)
```

Input

N	Number of elements of vector x.
ALPHA	The scalar ALPHA.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

X	The scaled array replaces the input.

Name F_SSORT/F_DSORT
Sort vector entries

Purpose F_xSORT sorts the entries of a real vector x in increasing or decreasing order and overwrites x with the sorted vector. If n is less than or equal to zero, F_xSORT returns immediately. F_xSORT is not defined for complex vectors; it operates strictly on real vectors.

Usage

```
INTEGER          INCX, N, SORT
REAL*4           X( * )
SUBROUTINE F_SSORT (SORT, N, X, INCX)

INTEGER          INCX, N, SORT
REAL*8           X( * )
SUBROUTINE F_DSORT (SORT, N, X, INCX)
```

Input

SORT	Specifies whether the data should be sorted in increasing or decreasing order. Use either **BLAS_INCREASING_ORDER** or **BLAS_DECREASING_ORDER**.
N	Number of elements of vector x.
X	REAL array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

X	The sorted array replaces the input.

Name

F_SSORTV/F_DSORTV
Sort vector and return index vector

Purpose

F_xSORTV sorts the entries of a real vector x in increasing or decreasing order, returns p, the permuted vector, and overwrites the vector x with the sorted vector ($x = P * x$). If n is less than or equal to zero, the routine returns immediately.

The permutation vector p represents a general permutation matrix P. This matrix P is represented as a product of at most n interchange permutations. An interchange permutation E is a permutation obtained by swapping two rows of the identity matrix. In other words, $P = E_n...E_1$ and each E_i is the identity with rows i and p_i interchanged.

F_xSORTV, like F_xSORT, strictly operates on real vectors and is not defined for complex vectors.

Usage

```
INTEGER         INCP, INCX, N, SORT
INTEGER         P( * )
REAL*4          X( * )
SUBROUTINE F_SSORTV (SORT, N, X, INCX, P, INCP)

INTEGER         INCP, INCX, N, SORT
INTEGER         P( * )
REAL*8          X( * )
SUBROUTINE F_DSORTV (SORT, N, X, INCX, P, INCP)
```

Input

SORT Specifies whether the data should be sorted in increasing or decreasing order. Use either **BLAS_INCREASING_ORDER** or **BLAS_DECREASING_ORDER**.

N Number of elements of vector x.

X REAL array, minimum length (**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

INCP Increment for the array p. A vector p having component p_i, i = 1,..., n, is stored in an array **P**() with increment argument **incp**. If **incp** > 0 then p_i is stored in **P** (1 + (i - 1) x **incp**). If **incp** < 0 then p_i is stored in **P** (1 + (**N** - i) x |**incp**|). **incp** = 0 is an illegal value.

Output	**X**	Updated (sorted) array replaces the input.
	P	Permuted vector. Array of integers, minimum length (**N** - 1) x |**incp**| + 1.

Name F_SSUM/F_DSUM/F_CSUM/F_ZSUM
Sum of entries of a vector

Purpose F_xSUM computes the sum of the entries of a vector x. If n is less than or equal to zero, F_xSUM returns immediately with the output scalar r set to zero.

$$r \leftarrow \sum_{i=0}^{n-1} x_i$$

Usage

```
INTEGER      INCX, N
REAL*4       X( * )
REAL*4       FUNCTION F_SSUM (N, X, INCX)

INTEGER      INCX, N
REAL*8       X( * )
REAL*8       FUNCTION F_DSUM (N, X, INCX)

INTEGER      INCX, N
COMPLEX*8    X( * )
COMPLEX*8    FUNCTION F_CSUM (N, X, INCX)

INTEGER      INCX, N
COMPLEX*16   X( * )
COMPLEX*16   FUNCTION F_ZSUM (N, X, INCX)
```

Input

N	Number of elements of vector x.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

Output

R	REAL scalar. The sum of the entries of vector x.

Name F_SSUMSQ/F_DSUMSQ/F_CSUMSQ/F_ZSUMSQ
Sum of squares

Purpose F_xSUMSQ returns the values *scl* and *ssq* such that

$$scl^2 \times ssq = scale^2 \times sumsq + \sum_{i=0}^{n-1} (Re(x_i)^2 + Im(x_i)^2)$$

The value of *sumsq* should be at least unity and the value of *ssq* then satisfies
$1.0 \le ssq \le (sumsq + n)$ when x is a real vector, and
$1.0 \le ssq \le (sumsq + 2n)$ when x is a complex vector.

scale should be non negative and *scl* returns the value

$$scl = max_{0 \le i < n}(scale, abs((Re(x_i)), abs(Im(x_i)))$$

Specify *scale* and *sumsq* on entry in **scl** and **ssq** respectively. **scl** and **ssq** are overwritten by **scl** and **ssq** respectively. The arguments **scl** and ssq are therefore always real scalars. If **n** is less than or equal to zero, the routine returns immediately with **scl** and **ssq** unchanged.

Usage

```
INTEGER          INCX, N
REAL*4           SCL, SSQ, X( * )
SUBROUTINE F_SSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER          INCX, N
REAL*8           SCL, SSQ, X( * )
SUBROUTINE F_DSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER          INCX, N
REAL*4           SCL, SSQ
COMPLEX*8        X( * )
SUBROUTINE F_CSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER          INCX, N
REAL*8           SCL, SSQ
COMPLEX*16       X( * )
SUBROUTINE F_ZSUMSQ (N, X, INCX, SSQ, SCL)
```

Input	**N**	Number of elements of vector x.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	SSQ	REAL scalar—inputs *sumsq*. If *sumsq* < 1, an error flag is set and passed to the error handler.
	SCL	REAL scalar—inputs *scale*. If *scale* < 0, an error flag is set and passed to the error handler.
Output	**SSQ**	REAL scalar replaces the input **ssq**.
	SCL	REAL scalar replaces the input **scl**.

Name F_SSWAP/F_DSWAP/F_CSWAP/F_ZSWAP
Interchange vectors

Purpose F_xSWAP interchanges the vectors x and y, that is, $x \leftrightarrow y$.

If n is less than or equal to zero, the routine returns immediately.

Usage

```
INTEGER        INCX, INCY, N
REAL*4         X( * ), Y( * )
SUBROUTINE F_SSWAP (N, X, INCX, Y, INCY)

INTEGER        INCX, INCY, N
REAL*8         X( * ), Y( * )
SUBROUTINE F_DSWAP (N, X, INCX, Y, INCY)

INTEGER        INCX, INCY, N
COMPLEX*8      X( * ), Y( * )
SUBROUTINE F_CSWAP (N, X, INCX, Y, INCY)

INTEGER        INCX, INCY, N
COMPLEX*16     X( * ), Y( * )
SUBROUTINE F_ZSWAP (N, X, INCX, Y, INCY)
```

Input

N	Number of elements of vector x.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.

Name F_SWAXPBY/F_DWAXPBY/F_CWAXPBY/F_ZWAXPBY
Scaled vector addition

Purpose F_xWAXPBY scales the vector x by α and the vector y by β, adds these two vectors, and stores the result in the vector w. If n is less than or equal to zero the routine returns immediately.

$$w \leftarrow \alpha x + \beta y$$

Usage

```
INTEGER          INCW, INCX, INCY, N
REAL*4           ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_SWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER          INCW, INCX, INCY, N
REAL*8           ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_DWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER          INCW, INCX, INCY, N
COMPLEX*8        ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_CWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER          INCW, INCX, INCY, N
COMPLEX*16       ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_ZWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)
```

Input

N	Number of elements of vector x.
ALPHA	The scalar ALPHA.
X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
BETA	The scalar BETA.

	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	INCW	Increment for the array *w*. A vector *w* having component w_i, i = 1,..., *n*, is stored in an array **W**() with increment argument **incw**. If **incw** > 0 then w_i is stored in **W** (1 + (i - 1) x **incw**). If **incw** < 0 then w_i is stored in **W**(1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
Output	**W**	The result of the addition of the two scaled vectors, *x* and *y*. REAL or COMPLEX array, minimum length (**N** - 1) x |**incw**| + 1.

3 Basic Matrix Operations

Overview

This chapter describes the subprograms in the Level 2 (two-loop) Basic Linear Algebra Subprograms (BLAS) and the Level 3 (three-loop) BLAS. Collectively, these two sets of subprograms are called the Extended BLAS.

This chapter also describes subprograms from the BLAS Standard. BLAS standardization efforts, supported by software and hardware vendors and university groups, began with a BLAS Technical (BLAST) Forum meeting in November 1995 at the University of Tennessee. The efforts of the BLAST Forum resulted in a BLAS Standard specification in 1999.

This chapter describes the subset of BLAS Standard matrix operations that are supported in HP MLIB 7.0. Refer to "BLAS Standard routines" on page 259.

This chapter explains how to use VECLIB matrix subprograms, which perform common computationally-intensive linear algebra operations. The operations described for both the legacy BLAS and BLAS Standard are:

- Basic matrix-vector operations
- Basic matrix-matrix operations

The most important of the Extended BLAS and BLAS Standard subprograms have been coded in highly-tuned assembly language.

Chapter objectives

After reading this chapter you will:

- Be familiar with the Extended BLAS subroutine naming convention
- Know what operations the Extended BLAS performs
- Know how to use the described subprograms
- Be familiar with the BLAS Standard subroutines supported in HP VECLIB

Associated documentation

The following documents provide supplemental material for this chapter:

Dongarra, J.J., J. DuCroz, S. Hammarling, and R. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. March, 1988. Vol. 14, No. 1.

Dongarra, J.J., J. DuCroz, S. Hammarling, and I. Duff. "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. March, 1990. Vol. 16, No. 1.

Higham, Nicholas J. "Is Fast Matrix Multiplication of Practical Use?" *SIAM News*. November, 1990. Vol. 23, No. 6.

What you need to know to use these subprograms

The following sections describe overall considerations for using matrix subprograms:

- Subroutine naming convention
- Operator arguments in the BLAS Standard

Subroutine naming convention

The Extended BLAS uses a subroutine naming convention that encodes the function of each subroutine into its name. Extended BLAS subprogram names consist of four, five, or six characters in the form TXXY, TXXYY, or TXXYYY.

The BLAS Standard uses the same naming convention as the Extended BLAS, with the addition of *F_* at the beginning of each routine name. That is, BLAS Standard subprogram names take the form F_TXXY, F_TXXYY, or F_TXXYYY.

For example, the legacy BLAS single-precision, triangular-solve routine is named STRSM and its BLAS Standard counterpart is named F_STRSM. Refer to "Legacy BLAS routines" on page 159 and "BLAS Standard routines" on page 259.

The first letter, denoted by T, in the naming convention indicates one of the four Fortran data types, as shown in Table 3-1.

Table 3-1 Extended BLAS Naming Convention—Data Type

T	Data Type
S	Single Precision REAL
D	Double Precision REAL
C	Single Precision COMPLEX
Z	Double Precision COMPLEX

The next two letters in the naming convention indicate the form of the matrix, as presented in Table 3-2.

Table 3-2 Extended BLAS Naming Convention—Matrix Form

XX	Form of Matrix
GE	General
GB	General band
HE	Hermitian
HB	Hermitian band
HP	Hermitian packed
SY	Symmetric
SB	Symmetric band
SP	Symmetric packed
TR	Triangular
TB	Triangular band
TP	Triangular packed

Table 3-3 lists the final one, two, or three characters in the naming convention, indicating the computation of a particular subroutine.

Table 3-3 Extended BLAS Naming Convention—Computation

YY	Subroutine Computation
MM	Matrix-Matrix multiply
MV	Matrix-Vector multiply
R	Rank-1 update
R2	Rank-2 update
RK	Rank-k update
R2K	Rank-2k update
SM	Solve multiple systems of linear equations
SV	Solve a system of linear equations

For example, SGBMV multiplies a vector (MV) by a general band matrix (GB) using the single precision REAL data type (S). ZTRSM solves a system of linear equations with one triangular coefficient matrix and a matrix of right-hand sides, using the double precision COMPLEX data type.

What you need to know to use these subprograms

The following sections describe overall considerations for using matrix subprograms:

- Subroutine naming convention
- Operator arguments in the BLAS Standard

Subroutine naming convention

The Extended BLAS uses a subroutine naming convention that encodes the function of each subroutine into its name. Extended BLAS subprogram names consist of four, five, or six characters in the form TXXY, TXXYY, or TXXYYY.

The BLAS Standard uses the same naming convention as the Extended BLAS, with the addition of *F_* at the beginning of each routine name. That is, BLAS Standard subprogram names take the form F_TXXY, F_TXXYY, or F_TXXYYY.

For example, the legacy BLAS single-precision, triangular-solve routine is named STRSM and its BLAS Standard counterpart is named F_STRSM. Refer to "Legacy BLAS routines" on page 159 and "BLAS Standard routines" on page 259.

The first letter, denoted by T, in the naming convention indicates one of the four Fortran data types, as shown in Table 3-1.

Table 3-1 Extended BLAS Naming Convention—Data Type

T	Data Type
S	Single Precision REAL
D	Double Precision REAL
C	Single Precision COMPLEX
Z	Double Precision COMPLEX

The next two letters in the naming convention indicate the form of the matrix, as presented in Table 3-2.

Table 3-2 Extended BLAS Naming Convention—Matrix Form

XX	Form of Matrix
GE	General
GB	General band
HE	Hermitian
HB	Hermitian band
HP	Hermitian packed
SY	Symmetric
SB	Symmetric band
SP	Symmetric packed
TR	Triangular
TB	Triangular band
TP	Triangular packed

Table 3-3 lists the final one, two, or three characters in the naming convention, indicating the computation of a particular subroutine.

Table 3-3 Extended BLAS Naming Convention—Computation

YY	Subroutine Computation
MM	Matrix-Matrix multiply
MV	Matrix-Vector multiply
R	Rank-1 update
R2	Rank-2 update
RK	Rank-k update
R2K	Rank-2k update
SM	Solve multiple systems of linear equations
SV	Solve a system of linear equations

For example, SGBMV multiplies a vector (MV) by a general band matrix (GB) using the single precision REAL data type (S). ZTRSM solves a system of linear equations with one triangular coefficient matrix and a matrix of right-hand sides, using the double precision COMPLEX data type.

Table 3-4 shows the valid combinations of T, XX, and Y, YY, or YYY. Each line indicates the allowable T prefixes and Y, YY, or YYY suffixes for a particular root name XX.

Table 3-4 Extended BLAS Naming Convention—Subprogram Names

Valid T				XX	Valid Y, YY, or YYY							
S	D			GE	MM	MV	R					
		C	Z	GE	MM	MV					RC	RU
S	D	C	Z	GB		MV						
		C	Z	HE	MM	MV	R	R2	RK	R2K		
		C	Z	HB		MV						
		C	Z	HP		MV	R	R2				
S	D			SY	MM	MV	R	R2	RK	R2K		
		C	Z	SY	MM				RK	R2K		
S	D			SB		MV						
S	D			SP		MV	R	R2				
S	D	C	Z	TR	MM	MV					SM	SV
S	D	C	Z	TB		MV						SV
S	D	C	Z	TP		MV						SV

The subprograms SGEMMS, DGEMMS, CGEMMS, and ZGEMMS, although not part of the standard Extended BLAS, are consistent with this nomenclature.

Operator arguments in the BLAS Standard

Some routines in the BLAS Standard take input-only arguments called operators. Operators allow for the specification of multiple related operations to be performed by a single function. The BLAS Standard specifies the type and the valid values these arguments should have according to the specific programming language.

Operator arguments used by the BLAS Standard routines are **NORM, SORT, SIDE, UPLO, TRANS, CONJ, DIAG**, and **JROT**. Refer to "Operator arguments" on page 13 for explanations of the valid operator values.

In BLAS Standard routines, you specify an operator argument with a named constant value. The actual numeric value assigned to the named constant is defined in the appropriate language's include file. Operator arguments are represented in the Fortran 77 interface as INTEGERs. This specification is different from the legacy BLAS, where operator arguments are defined as CHARACTER*1.

Refer to individual routines in "BLAS Standard routines" on page 259 for the named constants you can use to specify operator arguments for basic matrix subprograms.

Subprograms for basic matrix operations

The following sections in this chapter describe the legacy Extended BLAS and BLAS Standard matrix subprograms included with VECLIB:

- Legacy BLAS routines
- BLAS Standard routines

Note that the specification for operator arguments is different in legacy BLAS routines than in BLAS Standard routines. Operator arguments are represented in the BLAS Standard Fortran 77 interface as INTEGERs; in the legacy BLAS they are defined as CHARACTER*1.

In BLAS Standard routines, you specify an operator argument with a named constant value. Refer to the individual routines in "BLAS Standard routines" on page 259 for the named constants you can use to specify operator arguments. The actual numeric value assigned to the named constant is defined in the f77blas.h include file.

Legacy BLAS routines

Name

SGBMV/DGBMV/CGBMV/ZGBMV
Matrix-vector multiply

Purpose

These subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A is an m-by-n band matrix stored in a two-dimensional array, A^T is the transpose of A, and A^* is the conjugate transpose of A.

A band matrix is a matrix whose nonzero elements all are near the principal diagonal. Specifically, $a_{ij} = 0$ if $i-j > kl$ or $j-i > ku$ for some integers kl and ku. The smallest such kl and ku for a given matrix are called the lower and upper bandwidths, respectively, and $k = kl+ku+1$ is the total bandwidth.

The product can be stored in the result array or optionally added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute matrix-vector products of the forms

$$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y, \text{ and } y \leftarrow \alpha A^* x + \beta y.$$

Refer to "F_SGBMV/F_DGBMV/F_CGBMV/F_ZGBMV" on page 274 for a description of the BLAS Standard subprograms for general matrix-vector multiply.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, you need only provide the elements within the band of A. The subprograms for general band matrices use less storage than the subprograms for general full matrices if $kl+ku < n$.

The following example illustrates the storage of general band matrices.

Consider the following matrix A of size $m = 9$ by $n = 8$, with lower and upper bandwidths $kl = 2$ and $ku = 3$, respectively:

11	12	13	14	0	0	0	0
21	22	23	24	25	0	0	0
31	32	33	34	35	36	0	0
0	42	43	44	45	46	47	0
0	0	53	54	55	56	57	58
0	0	0	64	65	66	67	68
0	0	0	0	75	76	77	78
0	0	0	0	0	86	87	88
0	0	0	0	0	0	97	98

A is given in an array **ab** with at least $kl+ku+1 = 6$rows and $n = 8$ columns as follows:

*	*	*	14	25	36	47	58
*	*	13	24	35	46	57	68
*	12	23	34	45	56	67	78
11	22	33	44	55	66	77	88
21	32	43	54	65	76	87	98
31	42	53	64	75	86	97	*

The asterisks in the ku-by-ku triangle at the upper left corner and in the $(kl+n-m)$-by-$(kl+n-m)$ triangle at the lower right corner represent elements of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, then it is stored in **ab**$(ku+1+i-j,j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, such that the principal diagonal is stored in row $ku+1$ of **ab**.

Usage

```
CHARACTER*1   trans
INTEGER*4     m, n, kl, ku, ldab, incx, incy
REAL*4        alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL SGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1   trans
INTEGER*4     m, n, kl, ku, ldab, incx, incy
REAL*8        alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL DGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, kl, ku, ldab, incx, incy
COMPLEX*8      alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1    trans
INTEGER*4      m, n, kl, ku, ldab, incx, incy
COMPLEX*16     alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL ZGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
```

Input

trans Transposition option for A:

'N' or 'n'	Compute $y \leftarrow \alpha Ax + \beta y$
'T' or 't'	Compute $y \leftarrow \alpha A^T x + \beta y$
'C' or 'c'	Compute $y \leftarrow \alpha A^* x + \beta y$

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix A, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **ab**, **x**, or **y**.

n Number of columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ab**, **x**, or **y**.

kl The lower bandwidth of A, that is, the number of nonzero diagonals below the principal diagonal in the band, $0 \leq \mathbf{kl} < \mathbf{n}$.

ku The upper bandwidth of A, that is, the number of nonzero diagonals above the principal diagonal in the band, $0 \leq \mathbf{ku} < \mathbf{n}$.

alpha The scalar α. If **alpha** = 0, the subprograms compute $y \leftarrow \beta y$ without referencing **ab** or **x**.

ab Array containing the m-by-n band matrix A in the compressed form described above. If a_{ij} is in the band, it is stored in **ab**(**ku**+1+i–j,j). The columns of A are stored in the columns of **ab**, and the diagonals of A are stored in rows 1 through **kl**+**ku**+1.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **kl**+**ku**+1.

x Array containing the vector x. The number of elements of x and the value of **lenx**, the dimension of the array **x**, depend on **trans**:

'N' or 'n'	x has n elements	**lenx** = (**n**−1)×|**incx**|+1
otherwise	x has m elements	**lenx** = (**m**−1)×|**incx**|+1

incx Increment for the array **x**, **incx** ≠ 0:

incx > 0 x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).

incx < 0 x is stored backward in array **x**; that is, if **trans** = 'N' or 'n', then x_i is stored in **x**((*i*−**n**)×**incx**+1); otherwise, x_i is stored in **x**((*i*−**m**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

y Array containing the vector y. The number of elements of y and the value of **leny**, the dimension of the array **y**, depend on **trans**:

'N' or 'n'	y has m elements	**leny** = (**m**−1)×|**incy**|+1
otherwise	y has n elements	**leny** = (**n**−1)×|**incy**|+1

Not used as input if **beta** = 0.

incy Increment for the array **y**, **incy** ≠ 0:

incy > 0 y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).

incy < 0 y is stored backward in array **y**; that is, if **trans** = 'N' or 'n', then y_i is stored in **y**((*i*−**m**)×**incy**+1); otherwise, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output **y** The updated y vector replaces the input.

Notes These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
m < 0
n < 0
kl < 0
ku < 0
ldab < **kl**+**ku**+1
incx = 0
incy = 0

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9 by 6 real band matrix whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in an array AB whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 TRANS
INTEGER*4   M,N,KL,KU,LDAB,INCX,INCY
REAL*4      ALPHA,BETA,AB(10,10),X(10),Y(10)
TRANS = 'N'
M = 9
N = 6
KL = 2
KU = 3
ALPHA = 1.0
BETA = 0.0
LDAB = 10
INCX = 1
INCY = 1
CALL SGBMV (TRANS,M,N,KL,KU,ALPHA,AB,LDAB,X,INCX,BETA,Y,INCY)
```

Example 2 Form the REAL*8 matrix-vector product $y = \frac{1}{2}y - \rho A^T x$, where ρ is a real scalar, A is a 6-by-9 real band matrix whose lower bandwidth is 1 and whose upper bandwidth is 2. A is stored in an array AB whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,KL,KU,LDAB
REAL*8    RHO,AB(10,10),X(10),Y(10)
M = 9
N = 6
KL = 1
KU = 2
LDAB = 10
CALL DGBMV ('TRANSPOSE',M,N,KL,KU,-RHO,AB,LDAB,X,1,0.5D0,Y,1)
```

Name

SGECPY/DGECPY/CGECPY/ZGECPY
Copy general matrix

Purpose

These subprograms copy the general matrix A to B, where A and B are m-by-n matrices. Optionally, A^T or A^* can be copied to the n-by-m matrix B.

Refer to "F_SGE_COPY/F_DGE_COPY/F_CGE_COPY/F_ZGE_COPY" on page 277 for a description of the BLAS Standard subprograms for general matrix copy.

Usage

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, ldb
REAL*4         a(lda, *), b(ldb, *)
CALL SGECPY(trans, m, n, a, lda, b, ldb)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, ldb
REAL*8         a(lda, *), b(ldb, *)
CALL DGECPY(trans, m, n, a, lda, b, ldb)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, ldb
COMPLEX*8      a(lda, *), b(ldb, *)
CALL CGECPY(trans, m, n, a, lda, b, ldb)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, ldb
COMPLEX*16     a(lda, *), b(ldb, *)
CALL ZGECPY(trans, m, n, a, lda, b, ldb)
```

Input

trans Transposition option:

trans = 'N' or 'n'	Copy A to B
trans = 'T' or 't'	Copy A^T to B
trans = 'C' or 'c'	Copy A^* to B

In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix A, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a** or **b**.

n Number of columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **b**.

a Array containing the m-by-n matrix A.

	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (m,1).
	b	Array containing the matrix B, whose size is indicated by **trans**: trans = 'N' or 'n' B is an m-by-n matrix otherwise B is an n-by-m matrix
	ldb	The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max (the number of rows of B,1).
Output	**c**	The updated C matrix replaces the input.

Notes

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (see xerbla(3m)) can be replaced with a user-supplied version to change the error procedure. Error conditions are

trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
m < 0
n < 0
lda < max(**m**,1)
ldb too small

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NON-TRANSPOSED' for 'N', 'TRANSPOSED' for 'T', or 'CONJUGATE-TRANSPOSED' for 'C'.

Name

SGEMM/DGEMM/CGEMM/ZGEMM
Matrix-matrix multiply

Purpose

These subprograms compute the matrix-matrix product AB, where A is an m-by-k matrix, and B is a k-by-n matrix. Optionally, A can be replaced by A^T or A^*, where A is a k-by-m matrix, and B can be replaced by B^T or B^*, where B is an n-by-k matrix. Here, A^T and B^T are the transposes and A^* and B^* are the conjugate-transposes of A and B, respectively. The product can be stored in the result matrix (which is always of size m-by-n) or optionally can be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix product and the result matrix. Specifically, these subprograms compute matrix products of the forms:

$$C \leftarrow \alpha AB + \beta C, \quad C \leftarrow \alpha A^T B + \beta C, \quad C \leftarrow \alpha A^* B + \beta C,$$
$$C \leftarrow \alpha AB^T + \beta C, \quad C \leftarrow \alpha A^T B^T + \beta C, \quad C \leftarrow \alpha A^* B^T + \beta C,$$
$$C \leftarrow \alpha AB^* + \beta C, \quad C \leftarrow \alpha A^T B^* + \beta C, \quad C \leftarrow \alpha A^* B^* + \beta C.$$

Refer to "F_SGEMM/F_DGEMM/F_CGEMM/F_ZGEMM" on page 280 for a description of the BLAS Standard subprograms for general matrix-matrix multiply.

Usage

```
CHARACTER*1   transa, transb
INTEGER*4     m, n, k, lda, ldb, ldc
REAL*4        alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL SGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   transa, transb
INTEGER*4     m, n, k, lda, ldb, ldc
REAL*8        alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   transa, transb
INTEGER*4     m, n, k, lda, ldb, ldc
COMPLEX*8     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   transa, transb
INTEGER*4     m, n, k, lda, ldb, ldc
COMPLEX*16    alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Input

transa Transposition option for A:

'N' or 'n'	Use m-by-k matrix A
'T' or 't'	Use A^T where A is a k-by-m matrix
'C' or 'c'	Use A^* where A is a k-by-m matrix

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

transb Transposition option for B:

'N' or 'n'	Use k-by-n matrix B
'T' or 't'	Use B^T where B is an n-by-k matrix
'C' or 'c'	Use B^* where B is an n-by-k matrix

where B^T is the transpose of B and B^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix C, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix C, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **b**, or **c**.

k The *middle* dimension of the matrix multiply, $\mathbf{k} \geq 0$. If $\mathbf{k} = 0$, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array containing the matrix A, whose size is indicated by **transa**:

'N' or 'n'	A is an m-by-k matrix
otherwise	A is a k-by-m matrix

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (the number of rows of A,1).

b Array containing the matrix B, whose size is indicated by **transb**:

'N' or 'n'	B is a k-by-n matrix
otherwise	B is an n-by-k matrix

ldb The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max (the number of rows of B,1).

	beta	The scalar β.
	c	Array containing the m-by-n matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**m**,1).
Output	**c**	The updated C matrix replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
transb ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
m < 0
n < 0
k < 0
lda too small
ldb too small
ldc < max(**m**,1)

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **transa** and **transb** arguments as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix product $C = AB$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C, also of dimension 10 by 10.

```
CHARACTER*1 TRANSA,TRANSB
INTEGER*4   M,N,K,LDA,LDB,LDC
REAL*4      ALPHA,BETA,A(10,10),B(10,10),C(10,10)
TRANSA = 'N'
TRANSB = 'N'
M = 9
N = 8
K = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
LDB = 10
LDC = 10
CALL SGEMM (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

Example 2 Form the REAL*8 matrix product $C = \frac{1}{2}C + \rho A^T B$, where ρ is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C, also of dimension 10 by 10.

```
INTEGER*4 M,N,K,LDA,LDB,LDC
REAL*8    RHO,A(10,10),B(10,10),C(10,10)
M = 9
N = 8
K = 6
LDA = 10
LDB = 10
LDC = 10
CALL DGEMM ('TRAN','NONTRAN',M,N,K,RHO,A,LDA,B,LDB,0.5D0,C,LDC)
```

Name

DGEMMS/ZGEMMS
Strassen matrix-matrix multiply

Purpose

These subprograms use Strassen's method to compute the matrix-matrix product AB, where A is an m-by-k matrix and B is a k-by-n matrix. Strassen's method is an algorithm for matrix multiplication that, under certain circumstances, uses fewer than mnk multiplications and additions. These subprograms have argument lists identical to the standard Level 3 BLAS subprograms DGEMM and ZGEMM in VECLIB. So to convert a program to call a Strassen subprogram instead of a standard matrix multiply, it is only necessary to change the subprogram name. With consistent upper or lower case coding, a simple preprocessor directive can select standard or Strassen matrix multiply calls. Work area management is done by the subprograms.

By using Strassen's method, these subprograms can be considerably faster than their VECLIB counterparts. Refer to "Notes" for details. In addition to computing the matrix-matrix product AB, A can be replaced by A^T or A^*, where A is a k-by-m matrix, and B can be replaced by B^T or B^*, where B is an n-by-k matrix. Here, A^T and B^T are the transposes and A^* and B^* are the conjugate-transposes of A and B, respectively. The product can be stored in the result matrix (which is always of size m-by-n) or optionally can be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix product and the result matrix. Specifically, these subprograms compute matrix products of the forms:

$$C \leftarrow \alpha AB + \beta C, \quad C \leftarrow \alpha A^TB + \beta C, \quad C \leftarrow \alpha A^*B + \beta C,$$
$$C \leftarrow \alpha AB^T + \beta C, \quad C \leftarrow \alpha A^TB^T + \beta C, \quad C \leftarrow \alpha A*B^T + \beta C,$$
$$C \leftarrow \alpha AB^* + \beta C, \quad C \leftarrow \alpha A^TB^* + \beta C, \quad C \leftarrow \alpha A^*B^* + \beta C.$$

Refer to "F_SGEMM/F_DGEMM/F_CGEMM/F_ZGEMM" on page 280 for a description of the equivalent BLAS Standard subprograms.

Usage

```
CHARACTER*1    transa, transb
INTEGER*4      m, n, k, lda, ldb, ldc
REAL*8         alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DGEMMS(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1    transa, transb
INTEGER*4      m, n, k, lda, ldb, ldc
COMPLEX*16     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZGEMMS(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Input

transa Transposition option for A:

'N' or 'n'	Use m-by-k matrix A
'T' or 't'	Use A^T where A is a k-by-m matrix
'C' or 'c'	Use A^* where A is a k-by-m matrix

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

transb Transposition option for B:

'N' or 'n'	Use k-by-n matrix B
'T' or 't'	Use B^T where B is an n-by-k matrix
'C' or 'c'	Use B^* where B is an n-by-k matrix

where B^T is the transpose of B and B^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix C, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix C, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **b**, or **c**.

k The *middle* dimension of the matrix multiply, **k** ≥ 0. If **k** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array containing the matrix A, whose size is indicated by **transa**:

'N' or 'n'	A is an m-by-k matrix
otherwise	A is a k-by-m matrix

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ max (the number of rows of A,1).

b Array containing the matrix B, whose size is indicated by **transb**:

'N' or 'n'	B is a k-by-n matrix
otherwise	B is an n-by-k matrix

ldb The leading dimension of array **b** as declared in the calling program unit, with **ldb** $\geq$ max (the number of rows of B,1).

	beta	The scalar β.
	c	Array containing the m-by-n matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**m**,1).
Output	**c**	The updated C matrix replaces the input.

Notes

Except for the extra character in the subprogram name, these subprograms conform to specifications of the Level 3 BLAS subprograms DGEMM and ZGEMM.

Because of their use of Strassen's method DGEMMS and ZGEMMS are asymptotically faster than standard matrix multiply methods such as those employed in the standard routines DGEMM and ZGEMM. In practice, these particular implementations are faster than their standard counterparts if $\min(m,n,k) > 700$ for ZGEMMS, or $\min(m,n,k) > 1500$ for DGEMMS. The speedup in the complex case is much more pronounced. That is due in large part to the complex bilinear reduction technique (implemented underneath Strassen's method) that allows two complex matrices to be multiplied using only 3/4 of the multiplications required by the traditional method. Also, the relative cost of data motion is lower in the complex case. The gains in the real case are marginal until n becomes very large.

In the operator norm, Strassen's method is slightly less stable than traditional matrix multiplication, and the computation of individual elements is unstable. The emerging consensus seems to be that Strassen's method is sufficiently stable for most applications. Partly for stability reasons, however, only 64-bit Strassen subprograms are available at this time.

For a good overview and bibliography of this subject, see Higham.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
transb ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
m < 0
n < 0
k < 0
lda too small
ldb too small
ldc < max(**m**,1)

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved, for example, by coding the **transa** and **transb** arguments as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*8 matrix product $C = AB$, where A is a 900-by-600 real matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 real matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 real matrix stored in an array C, also of dimension 1000 by 1000.

```
      CHARACTER*1 TRANSA,TRANSB
      INTEGER*4   M,N,K,LDA,LDB,LDC
      REAL*8      ALPHA,BETA,A(1000,1000),B(1000,1000),
     &            C(1000,1000)
      TRANSA = 'N'
      TRANSB = 'N'
      M = 900
      N = 800
      K = 600
      ALPHA = 1.0
      BETA = 0.0
      LDA = 1000
      LDB = 1000
      LDC = 1000
      CALL DGEMMS (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,
     &             BETA,C,LDC)
```

Example 2 Form the COMPLEX*16 matrix product $C = \frac{1}{2}C + \rho A*B$, where ρ is a complex scalar, A is a 600-by-900 complex matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 complex matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 complex matrix stored in an array C, also of dimension 1000 by 1000.

```
      INTEGER*4   M,N,K,LDA,LDB,LDC
      COMPLEX*16 RHO,A(1000,1000),B(1000,1000),C(1000,1000)
      M = 900
      N = 800
      K = 600
      LDA = 1000
      LDB = 1000
      LDC = 1000
      CALL ZGEMMS ('CONJ','NORMAL',M,N,K,RHO,A,LDA,B,LDB,
     &             (0.5D0,0.0D0),C,LDC)
```

Name

SGEMV/DGEMV/CGEMV/ZGEMV
Matrix-vector multiply

Purpose

These subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A is an m-by-n matrix, A^T is the transpose of A, and A^* is the conjugate transpose of A. The product can be stored in the result array or added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute matrix-vector products of the forms:

$$y \leftarrow aAx + \beta y,\ y \leftarrow aA^Tx + \beta y,\ \text{and}\ y \leftarrow aA^*x + \beta y.$$

Refer to "F_SGEMV/F_DGEMV/F_CGEMV/F_ZGEMV" on page 283 for a description of the BLAS Standard subprograms for a triangular matrix-vector multiply.

Usage

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, incx, incy
REAL*4         alpha, beta, a(lda, n), x(lenx), y(leny)
CALL SGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, incx, incy
REAL*8         alpha, beta, a(lda, n), x(lenx), y(leny)
CALL DGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, incx, incy
COMPLEX*8      alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1    trans
INTEGER*4      m, n, lda, incx, incy
COMPLEX*16     alpha, beta, a(lda, n), x(lenx), y(leny)
CALL ZGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

Input

trans Transposition option for A:

'N' or 'n'	Compute $y \leftarrow \alpha Ax + \beta y$
'T' or 't'	Compute $y \leftarrow \alpha A^T x + \beta y$
'C' or 'c'	Compute $y \leftarrow \alpha A^* x + \beta y$

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix A, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a**, **x**, or **y**.

n Number of columns in matrix A, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $y \leftarrow \beta y$ without referencing **A** or **x**.

a Array containing the m-by-n matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**m**,1).

x Array containing the vector x. The number of elements of x and the value of **lenx**, the dimension of the array **x**, depend on **trans**:

'N' or 'n'	x has n elements	**lenx** = (**n**−1)×|**incx**|+1
otherwise	x has m elements	**lenx** = (**m**−1)×|**incx**|+1

incx Increment for the array **x**, **incx** ≠ 0:

incx > 0 x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).

incx < 0 x is stored backward in array **x**; that is, if **trans** = 'N' or 'n', then x_i is stored in **x**((*i*−**n**)×**incx**+1); otherwise, x_i is stored in **x**((*i*−**m**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

y Array containing the vector y. The number of elements of y and the value of **leny**, the dimension of the array **y**, depend on **trans**:

'N' or 'n'	y has m elements	**leny** = (**m**−1)×|**incy**|+1
otherwise	y has n elements	**leny** = (**n**−1)×|**incy**|+1

Not used as input if **beta** = 0.

incy Increment for the array **y**, **incy** ≠ 0:

incy > 0 y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).

incy < 0 y is stored backward in array **y**; that is, if **trans** = 'N' or 'n', then y_i is stored in **y**((*i*−**m**)×**incy**+1); otherwise, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

y The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
m < 0
n < 0
lda < max(**m**,1)
incx = 0
incy = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 10-by-10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 TRANS
INTEGER*4   M,N,LDA,INCX,INCY
REAL*4      ALPHA,BETA,A(10,10),X(10),Y(10)
TRANS = 'N'
M = 9
N = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
INCX = 1
INCY = 1
CALL SGEMV (TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

Example 2 Form the REAL*8 matrix-vector product $y = \frac{1}{2}y - \rho A^T x$, where ρ is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10-by-10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,LDA
REAL*8    RHO,A(10,10),X(10),Y(10)
M = 9
N = 6
LDA = 10
CALL DGEMV ('TRANSPOSE',M,N,-RHO,A,LDA,X,1,0.5D0,Y,1)
```

Name

SGER/DGER/CGERC/CGERU/ZGERC/ZGERU
Rank-1 update

Purpose

These subprograms compute the rank-1 updates

$$A \leftarrow axy^T + A \quad \text{and} \quad A \leftarrow axy^* + A,$$

where A is an m-by-n matrix, α is a scalar, x is an m-vector, y is an n-vector, and y^T and y^* are the transpose and conjugate transpose of y, respectively.

Refer to "F_SGER/F_DGER/F_CGER/F_ZGER" on page 290 for a description of the BLAS Standard subprograms for general rank-1 update.

Usage

```
INTEGER*4       m, n, lda, incx, incy
REAL*4          alpha, a(lda, n), x(lenx), y(leny)
CALL SGER(m, n, alpha, x, incx, y, incy, a, lda)

INTEGER*4       m, n, lda, incx, incy
REAL*8          alpha, a(lda, n), x(lenx), y(leny)
CALL DGER(m, n, alpha, x, incx, y, incy, a, lda)

INTEGER*4       m, n, lda, incx, incy
COMPLEX*8       alpha, a(lda, n), x(lenx), y(leny)
CALL CGERC(m, n, alpha, x, incx, y, incy, a, lda)

INTEGER*4       m, n, lda, incx, incy
COMPLEX*8       alpha, a(lda, n), x(lenx), y(leny)
CALL CGERU(m, n, alpha, x, incx, y, incy, a, lda)

INTEGER*4       m, n, lda, incx, incy
COMPLEX*16      alpha, a(lda, n), x(lenx), y(leny)
CALL ZGERC(m, n, alpha, x, incx, y, incy, a, lda)

INTEGER*4       m, n, lda, incx, incy
COMPLEX*16      alpha, a(lda, n), x(lenx), y(leny)
CALL ZGERU(m, n, alpha, x, incx, y, incy, a, lda)
```

Input

m	Number of rows in matrix A and elements of vector x, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a**, **x**, or **y**.
n	Number of columns in matrix A and elements of vector y, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.
alpha	The scalar α. If **alpha** = 0, the subprograms do not reference **A**, **x**, or **y**.

	x	Array of length **lenx** = (**m**−1)×|**incx**|+1 containing the m-vector x.
	incx	Increment for the array **x**, **incx** ≠ 0: **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((i−1)×**incx**+1). **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((i−**m**)×**incx**+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	y	Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. y is used in conjugated form by CGERC and ZGERC, and in unconjugated form by the other subprograms.
	incy	Increment for the array **y**, **incy** ≠ 0: **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((i−1)×**incy**+1). **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((i−**n**)×**incy**+1). Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	a	Array containing the **m**-by-**n** matrix A.
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**m**,1).
Output	**a**	The updated A matrix replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

m < 0
n < 0
lda < max(**m**,1)
incx = 0
incy = 0

Example 1 Apply a REAL*4 rank-1 update xy^T to A, where A is a 6-by-9 real matrix stored in an array A whose dimensions are 10-by-10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,LDA,INCX,INCY
REAL*4    ALPHA,A(10,10),X(10),Y(10)
M = 6
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
INCY = 1
CALL SGER (M,N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

Example 2 Apply a COMPLEX*8 conjugated rank-1 update $-2xy^*$ to A, where A is a 6-by-9 complex matrix stored in an array A whose dimensions are 10 by 10, x is a complex vector 6 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,LDA
COMPLEX*8 A(10,10),X(10),Y(10)
M = 6
N = 9
LDA = 10
CALL CGERC (M,N,(-2.0E0,0.0E0),X,1,Y,1,A,LDA)
```

Name

SSBMV/DSBMV/CHBMV/ZHBMV
Matrix-vector multiply

Purpose

These subprograms compute the matrix-vector product Ax, where A is an n-by-n real symmetric or complex Hermitian band matrix and x is a real or complex n-vector. The product can be stored in the result array, or it can be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute the matrix-vector product of the form:

$$y \leftarrow \alpha Ax + \beta y.$$

The structure of A is indicated by the name of the subprogram used:

SSBMV	or	DSBMV	A is a real symmetric band matrix
CHBMV	or	ZHBMV	A is a complex Hermitian band matrix

A symmetric or Hermitian band matrix is a symmetric or Hermitian matrix whose nonzero elements all are on, or fairly near, the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $|i-j| \leq kd$ for some integer kd, called the half bandwidth.

Refer to "F_SSBMV/F_DSBMV/F_CSBMV/F_ZSBMV" on page 292 and "F_CHBMV/F_ZHBMV" on page 259 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and because either triangle of A can be obtained from the other, you only need to provide the band within one triangle of A. Compared to storing the entire matrix, this can save memory in two ways: Only the elements within the band are stored and of them only the upper or the lower triangle.

The following examples illustrate the storage of symmetric band matrices. Consider the following matrix A of order $n = 7$ and half bandwidth $kd = 2$:

$$\begin{matrix} 11 & 12 & 13 & 0 & 0 & 0 & 0 \\ 12 & 22 & 23 & 24 & 0 & 0 & 0 \\ 13 & 23 & 33 & 34 & 35 & 0 & 0 \\ 0 & 24 & 34 & 44 & 45 & 46 & 0 \\ 0 & 0 & 35 & 45 & 55 & 56 & 57 \\ 0 & 0 & 0 & 46 & 56 & 66 & 67 \\ 0 & 0 & 0 & 0 & 57 & 67 & 77 \end{matrix}$$

Upper triangular storage

The upper triangle of A is stored in an array **ab** with at least $kd+1 = 3$ rows and 7 columns as follows:

*	*	13	24	35	46	57
*	12	23	34	45	56	67
11	22	33	44	55	66	77

The asterisks represent elements in the kd-by-kd triangle at the upper left corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of the upper triangle of A, it is stored in **ab**($kd+1+i-j,j$). Therefore, the columns of the upper triangle of A are stored in the columns of **ab**, and the diagonals of the upper triangle of A are stored in the rows of **ab**, with the principal diagonal in row $kd+1$, the first superdiagonal starting in the second position in row kd, and so on.

Lower triangular storage

The lower triangle of A is stored in the array **ab** as follows:

11	22	33	44	55	66	77
12	23	34	45	56	67	*
13	24	35	46	57	*	*

The asterisks represent elements in the kd-by-kd triangle at the lower right corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of the lower triangle of A, it is stored in **ab**($1+i-j,j$). Therefore, the columns of the lower triangle of A are stored in the columns of **ab**, and the diagonals of the lower triangle of A are stored in the rows of **ab**, with the principal diagonal in the first row, the first subdiagonal in the second row, and so on.

Usage

```
CHARACTER*1     uplo
INTEGER*4       n, kd, ldab, incx, incy
REAL*4          alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL SSBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1     uplo
INTEGER*4       n, kd, ldab, incx, incy
REAL*8          alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL DSBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)
```

```
CHARACTER*1    uplo
INTEGER*4      n, kd, ldab, incx, incy
COMPLEX*8      alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CHBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1    uplo
INTEGER*4      n, kd, ldab, incx, incy
COMPLEX*16     alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL ZHBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	The lower triangle of A is stored.
'U' or 'u'	The upper triangle of A is stored.

n Number of rows and columns in matrix A, **n** ≥ 0. If **n** = 0, the subprograms do not reference **ab** or **x**.

kd The number of nonzero diagonals above or below the principal diagonal.

alpha The scalar α. If **alpha** = 0, the subprograms compute $y \leftarrow \beta y$ without referencing **ab** or **x**.

ab Array containing the n-by-n symmetric band matrix A in the compressed form described above. The columns of the band of A are stored in the columns of **ab**, and the diagonals of the band of A are stored in the rows of **ab**.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** $\geq$ **kd**+1.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the input vector x.

incx Increment for the array **x**, **incx** $\neq 0$:

incx > 0	x is stored forward in array **x**; that is, x_i is stored in **x**((i−1)×**incx**+1).
incx < 0	x is stored backward in array **x**; that is, x_i is stored in **x**((i−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

	y	Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.
	incy	Increment for the array **y**, **incy** ≠ 0: **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((i−1)×**incy**+1). **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((i−**n**)×**incy**+1). Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**y**	The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
kd < 0
ldab < **kd**+1
incx = 0
incy = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 75-by-75 real symmetric band matrix with half bandwidth 15 whose lower triangular part is stored in an array AB whose dimensions are 25-by-100, and x and y are real vectors 75 elements long stored in arrays X and Y of dimension 100, respectively.

```
CHARACTER*1 UPLO
INTEGER*4   N,KD,LDAB,INCX,INCY
REAL*4      AB(25,100),X(100),Y(100)
UPLO = 'L'
N = 75
KD = 15
LDAB = 25
INCX = 1
INCY = 1
CALL SSBMV (UPLO, N, KD, 1.0, AB, LDAB, X, INCX, 0.0, Y, INCY)
```

Example 2 Form the REAL*8 matrix-vector product $y = Ax$, where A is a 75-by-75 real symmetric band matrix with half bandwidth 15 whose upper triangle is stored in an array AB whose dimensions are 25-by-100, and x and y are real vectors 75 elements long stored in arrays X and Y of dimension 100, respectively.

```
INTEGER*4 N,KD,LDAB
REAL*4    AB(25,100),X(100),Y(100)
N = 75
KD = 15
LDAB = 25
CALL DSBMV ('UPPER', N, KD, 1.0, AB, LDAB, X, 1, 1.0, Y, 1)
```

Name SSPMV/DSPMV/CHPMV/ZHPMV
Matrix-vector multiply

Purpose These subprograms compute the matrix-vector product Ax, where A is an n-by-n real symmetric or complex Hermitian matrix stored in packed form as described in "Matrix Storage," and x is a real or complex n-vector. The product can be stored in the result array, or, optionally, be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute the matrix-vector product of the form:

$$y \leftarrow \alpha Ax + \beta y.$$

The structure of A is indicated by the name of the subprogram used:

SSPMV	or	DSPMV	A is a real symmetric matrix
CHPMV	or	ZHPMV	A is a complex Hermitian matrix

Refer to "F_SSPMV/F_DSPMV/F_CSPMV/F_ZSPMV" on page 294 and "F_CHPMV/F_ZHPMV" on page 267 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array.

The following examples illustrate the packed storage of symmetric or Hermitian matrices.

Upper triangular storage

If the upper triangle of A is

$$\begin{matrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 24 \\ & & 33 & 34 \\ & & & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper triangular matrix element a_{ij} is stored in array element **ap**(i+(j×(j–1))/2).

Lower triangular storage

If the lower triangle of A is

$$\begin{matrix} 11 & & & \\ 21 & 22 & & \\ 31 & 32 & 33 & \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower triangular matrix element a_{ij} is stored in array element **ap**(i+((j–1)×(2n–j))/2).

Usage

```
CHARACTER*1     uplo
INTEGER*4       n, incx, incy
REAL*4          alpha, beta, ap(lenap), x(lenx), y(leny)
CALL SSPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx, incy
REAL*8          alpha, beta, ap(lenap), x(lenx), y(leny)
CALL DSPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx, incy
COMPLEX*8       alpha, beta, ap(lenap), x(lenx), y(leny)
CALL CHPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx, incy
COMPLEX*16      alpha, beta, ap(lenap), x(lenx), y(leny)
CALL ZHPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

Input

uplo Upper/lower triangular option for A:

- 'L' or 'l' — The lower triangle of A is stored in the packed array.
- 'U' or 'u' — The upper triangle of A is stored in the packed array.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ap**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $y \leftarrow \beta y$ without referencing **ap** or **x**.

ap Array of length **lenap** = **n**×(**n**+1)/2 containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

- **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output **y** The updated y vector replaces the input.

Notes These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
incx = 0
incy = 0

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array AP of dimension 55, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,INCX,INCY
REAL*4      ALPHA,BETA,AP(55),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
BETA = 0.0
INCX = 1
INCY = 1
CALL SSPMV (UPLO,N,ALPHA,AP,X,INCX,BETA,Y,INCY)
```

Example 2 Form the COMPLEX*8 matrix-vector product $y = \frac{1}{2}y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 N
COMPLEX*8 RHO,AP(55),X(10),Y(10)
N = 9
CALL CHPMV ('LOWER',N,-RHO,AP,X,1,(0.5E0,0.0E0),Y,1)
```

Name SSPR/DSPR/CHPR/ZHPR
Rank-1 update

Purpose These subprograms compute the real symmetric or complex Hermitian rank-1 update

$$A \leftarrow axx^* + A,$$

where A is an n-by-n real symmetric or complex Hermitian matrix stored in packed form as described in "Matrix Storage," α is a real scalar, x is a real or complex n-vector, and x^* is the conjugate transpose of x. (The conjugate transpose of a real vector is simply the transpose.)

The structure of A is indicated by the name of the subprogram used:

SSPR	or	DSPR	A is a real symmetric matrix
CHPR	or	ZHPR	A is a complex Hermitian matrix

Refer to "F_SSPR/F_DSPR/F_CSPR/F_ZSPR" on page 296, and "F_CHPR/F_ZHPR" on page 269 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array.

The following examples illustrate the packed storage of symmetric or Hermitian matrices.

Upper triangular storage

If the upper triangle of A is

$$\begin{matrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 24 \\ & & 33 & 34 \\ & & & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper triangular matrix element a_{ij} is stored in array element **ap**(i+(j×(j–1))/2).

Lower triangular storage

If the lower triangle of A is

$$\begin{matrix} 11 & & & \\ 21 & 22 & & \\ 31 & 32 & 33 & \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower triangular matrix element a_{ij} is stored in array element **ap**($i+((j-1)\times(2n-j))/2$).

Usage

```
CHARACTER*1     uplo
INTEGER*4       n, incx
REAL*4          alpha, ap(lenap), x(lenx)
CALL SSPR(uplo, n, alpha, x, incx, ap)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx
REAL*8          alpha, ap(lenap), x(lenx)
CALL DSPR(uplo, n, alpha, x, incx, ap)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx
REAL*4          alpha
COMPLEX*8       ap(lenap), x(lenx)
CALL CHPR(uplo, n, alpha, x, incx, ap)
```

```
CHARACTER*1     uplo
INTEGER*4       n, incx
REAL*8          alpha
COMPLEX*16      ap(lenap), x(lenx)
CALL ZHPR(uplo, n, alpha, x, incx, ap)
```

Input

uplo Upper/lower triangular option for A:

- 'L' or 'l' — The lower triangle of A is stored in the packed array.
- 'U' or 'u' — The upper triangle of A is stored in the packed array.

n Number of rows and columns in matrix A and elements of vector x, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ap** or **x**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **ap** or **x**.

x Array of length **lenx** = $(\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x.

incx Increment for the array **x**, **incx** $\neq 0$:

- **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in $\mathbf{x}((i-\mathbf{n})\times\mathbf{incx}+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

ap Array of length **lenap** = $\mathbf{n}\times(\mathbf{n}+1)/2$ containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

Output

ap The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1

Apply a REAL*4 symmetric rank-1 update xx^T to A, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array AP of dimension 55, and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,INCX
REAL*4      ALPHA,AP(55),X(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
INCX = 1
CALL SSPR (UPLO,N,ALPHA,X,INCX,AP)
```

Example 2

Apply a COMPLEX*8 Hermitian rank-1 update $-2xx^*$ to A, where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, and x is a complex vector 9 elements long stored in an array X of dimension 10.

```
INTEGER*4 N
COMPLEX*8 AP(55),X(10)
N = 9
CALL CHPR ('LOWER',N,-2.0,X,1,AP)
```

Name

SSPR2/DSPR2/CHPR2/ZHPR2
Rank-2 update

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-2 update

$$A \leftarrow \alpha xy^* + \bar{\alpha} yx^* + A,$$

where A is an n-by-n real symmetric or complex Hermitian matrix stored in packed form as described in "Matrix Storage," α is a complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are real or complex n-vectors, and x^* and y^* are the conjugate transposes of x and y, respectively. (The conjugate of a real scalar is just the scalar, and the conjugate transpose of a real vector is simply the transpose.)

The structure of A is indicated by the name of the subprogram used:

SSPR2	or	DSPR2	A is a real symmetric matrix
CHPR2	or	ZHPR2	A is a complex Hermitian matrix

Refer to "F_SSPR2/F_DSPR2/F_CSPR2/F_ZSPR2" on page 298 and "F_CHPR2/F_ZHPR2" on page 271 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array.

The following examples illustrate the packed storage of symmetric or Hermitian matrices.

Upper triangular storage

If the upper triangle of A is

$$\begin{matrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 24 \\ & & 33 & 34 \\ & & & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper triangular matrix element a_{ij} is stored in array element **ap**($i+(j\times(j-1))/2$).

Lower triangular storage

If the lower triangle of A is

$$\begin{matrix} 11 & & & \\ 21 & 22 & & \\ 31 & 32 & 33 & \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower triangular matrix element a_{ij} is stored in array element **ap**($i+((j-1)\times(2n-j))/2$).

Usage

```
CHARACTER*1   uplo
INTEGER*4     n, incx, incy
REAL*4        alpha, ap(lenap), x(lenx), y(leny)
CALL SSPR2(uplo, n, alpha, x, incx, y, incy, ap)
```

```
CHARACTER*1   uplo
INTEGER*4     n, incx, incy
REAL*8        alpha, ap(lenap), x(lenx), y(leny)
CALL DSPR2(uplo, n, alpha, x, incx, y, incy, ap)
```

```
CHARACTER*1   uplo
INTEGER*4     n, incx, incy
COMPLEX*8     alpha, ap(lenap), x(lenx), y(leny)
CALL CHPR2(uplo, n, alpha, x, incx, y, incy, ap)
```

```
CHARACTER*1   uplo
INTEGER*4     n, incx, incy
COMPLEX*16    alpha, ap(lenap), x(lenx), y(leny)
CALL ZHPR2(uplo, n, alpha, x, incx, y, incy, ap)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	The lower triangle of A is stored in the packed array.
'U' or 'u'	The upper triangle of A is stored in the packed array.

n Number of rows and columns in matrix A and elements of vectors x and y, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ap**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **ap**, **x**, or **y**.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

- **incx** > 0: x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0: x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0: y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0: y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

ap Array of length **lenap** = **n**×(**n**+1)/2 containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

Output

ap The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
incx = 0
incy = 0

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1

Apply a REAL*4 symmetric rank-2 update xy^T+x^Ty to A, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array AP of dimension 55, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,INCX,INCY
REAL*4      ALPHA,AP(55),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
INCX = 1
INCY = 1
CALL SSPR2 (UPLO,N,ALPHA,X,INCX,Y,INCY,AP)
```

Example 2

Apply a COMPLEX*8 Hermitian rank-2 update $\alpha xy^*+\bar{\alpha}yx^*$ to A, where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, α is a complex scalar, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y of dimension 10.

```
INTEGER*4 N
COMPLEX*8 ALPHA,AP(55),X(10),Y(10)
N = 9
CALL CHPR2 ('LOWER',N,ALPHA,X,1,Y,1,AP)
```

Name

SSYMM/DSYMM/CHEMM/CSYMM/ZHEMM/ZSYMM
Matrix-matrix multiply

Purpose

These subprograms compute the matrix-matrix products AB and BA, where A is a real symmetric, complex symmetric, or complex Hermitian matrix, and B is an m-by-n matrix. The size of A, either m-by-m or n-by-n, depends on which matrix product is requested. The product can be stored in the result matrix (which is always of size m-by-n) or, optionally, can be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix product and the result matrix. Specifically, these subprograms compute matrix products of the forms:

$$C \leftarrow \alpha AB + \beta C \quad \text{and} \quad C \leftarrow \alpha BA + \beta C.$$

The structure of A is indicated by the name of the subprogram used:

SSYMM	or	DSYMM	A is a real symmetric matrix
CHEMM	or	ZHEMM	A is a complex Hermitian matrix
CSYMM	or	ZSYMM	A is a complex symmetric matrix

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1   side, uplo
INTEGER*4     m, n, lda, ldb, ldc
REAL*4        alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL SSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   side, uplo
INTEGER*4     m, n, lda, ldb, ldc
REAL*8        alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   side, uplo
INTEGER*4     m, n, lda, ldb, ldc
COMPLEX*8     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CHEMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   side, uplo
INTEGER*4     m, n, lda, ldb, ldc
COMPLEX*8     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
CHARACTER*1    side, uplo
INTEGER*4      m, n, lda, ldb, ldc
COMPLEX*16     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZHEMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1    side, uplo
INTEGER*4      m, n, lda, ldb, ldc
COMPLEX*16     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
```

Input

side Specifies whether symmetric or Hermitian matrix A is the left or right matrix operand:

- 'L' or 'l' — A is the left matrix operand, that is, compute $C \leftarrow \alpha AB + \beta C$
- 'R' or 'r' — A is the right matrix operand, that is, compute $C \leftarrow \alpha BA + \beta C$

uplo Upper/lower triangular storage option for A:

- 'L' or 'l' — Reference only the lower triangle of A
- 'U' or 'u' — Reference only the upper triangle of A

m Number of rows in matrix C, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix B, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **b**, or **c**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the matrix A. The other triangle of **a** is not referenced. The size of A is indicated by **side**:

- 'L' or 'l' — A is m-by-m
- 'R' or 'r' — A is n-by-n

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ max (the number of rows of A,1).

b Array containing the m-by-n matrix B.

ldb The leading dimension of array **b** as declared in the calling program unit, with **ldb** $\geq$ max(**m**,1).

beta The scalar β.

	c	Array containing the m-by-n matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**m**,1).
Output	**c**	The updated C matrix replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

side ≠ 'L' or 'l' or 'R' or 'r'
uplo ≠ 'L' or 'l' or 'U' or 'u'
m < 0
n < 0
lda too small
ldb < max(**m**,1)
ldc < max(**m**,1)

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **side** argument as 'LEFT' for 'L' or 'RIGHT' for 'R'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix product $C = AB$, where A is a 6-by-6 real symmetric real matrix whose upper triangle is stored in the upper triangle of an array A of dimension 10-by-10, B is a 6-by-8 real matrix stored in an array B of dimension 10-by-10, and C is a 6-by-8 real matrix stored in an array C, also of dimension 10-by-10.

```
CHARACTER*1 SIDE,UPLO
INTEGER*4   M,N,LDA,LDB,LDC
REAL*4      ALPHA,BETA,A(10,10),B(10,10),C(10,10)
SIDE = 'L'
UPLO = 'U'
M = 6
N = 8
ALPHA = 1.0
BETA = 0.0
LDA = 10
LDB = 10
LDC = 10
CALL SSYMM (SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

Example 2 Form the COMPLEX*8 matrix-matrix product $C = \frac{1}{2}BA - \rho C$, where ρ is a scalar, A is an 8-by-8 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A of dimension 10-by-10, B is a 6-by-8 complex matrix stored in an array whose dimensions are 10-by-10, and C is a 6-by-8 complex matrix stored in an array C, also of dimension 10-by-10.

```
INTEGER*4 M,N,LDA,LDB,LDC
COMPLEX*8 HALF,RHO,A(10,10),B(10,10),C(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
LDC = 10
HALF = (0.5,0.0)
CALL CHEMM ('RIGHT','LOWER',M,N,-RHO,A,LDA,B,LDB,HALF,C,LDC)
```

Name

SSYMV/DSYMV/CHEMV/ZHEMV
Matrix-vector multiply

Purpose

These subprograms compute the matrix-vector product Ax, where A is an n-by-n real symmetric or complex Hermitian matrix, and x is a real or complex n-vector. The product can be stored in the result array, or, optionally, be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute the matrix-vector product of the form

$$y \leftarrow \alpha Ax + \beta y.$$

The structure of A is indicated by the name of the subprogram used:

SSYMV	or	DSYMV	A is a real symmetric matrix
CHEMV	or	ZHEMV	A is a complex Hermitian matrix

Refer to "F_SSYMV/F_DSYMV/F_CSYMV/F_ZSYMV" on page 300 and "F_CHEMV/F_ZHEMV" on page 261 for equivalent BLAS Standard subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1   uplo
INTEGER*4     n, lda, incx, incy
REAL*4        alpha, beta, a(lda, n), x(lenx), y(leny)
CALL SSYMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1   uplo
INTEGER*4     n, lda, incx, incy
REAL*8        alpha, beta, a(lda, n), x(lenx), y(leny)
CALL DSYMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1   uplo
INTEGER*4     n, lda, incx, incy
COMPLEX*8     alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CHEMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1   uplo
INTEGER*4     n, lda, incx, incy
COMPLEX*16    alpha, beta, a(lda, n), x(lenx), y(leny)
CALL ZHEMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	Reference only the lower triangle of A.
'U' or 'u'	Reference only the upper triangle of A.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $y \leftarrow \beta y$ without referencing **a** or **x**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of an n-by-n real symmetric or complex Hermitian matrix A. The other triangle of **a** is not referenced.

lda The leading dimension of array **a** as declared in the calling program unit, with $\mathbf{lda} \geq \max(\mathbf{n},1)$.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x.

incx Increment for the array **x**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$	x is stored forward in array **x**; that is, x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$	x is stored backward in array **x**; that is, x_i is stored in $\mathbf{x}((i-\mathbf{n})\times\mathbf{incx}+1)$.

Use $\mathbf{incx} = 1$ if the vector x is stored contiguously in array **x**, that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

y Array of length $\mathbf{leny} = (\mathbf{n}-1)\times|\mathbf{incy}|+1$ containing the n-vector y. Not used as input if $\mathbf{beta} = 0$.

incy Increment for the array **y**, **incy** ≠ 0:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((i–1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((i–**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

y The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
lda < max(**n**,1)
incx = 0
incy = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in the upper triangle of an array A whose dimensions are 10-by-10, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,LDA,INCX,INCY
REAL*4      ALPHA,BETA,A(10,10),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
BETA = 0.0
LDA = 10
INCX = 1
INCY = 1
CALL SSYMV (UPLO,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

Example 2 Form the COMPLEX*8 matrix-vector product $y = \frac{1}{2}y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10-by-10, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 N,LDA
COMPLEX*8 RHO,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHEMV ('LOWER',N,-RHO,A,LDA,X,1,(0.5,0.0),Y,1)
```

Name

SSYR/DSYR/CHER/ZHER
Rank-1 update

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-1 update

$$A \leftarrow \alpha x x^* + A,$$

where A is an n-by-n real symmetric or complex Hermitian matrix, α is a real scalar, x is a real or complex n-vector, and x^* is the conjugate transpose of x. (The conjugate transpose of a real vector is simply the transpose.)

The structure of A is indicated by the name of the subprogram used:

SSYR	or	DSYR	A is a real symmetric matrix
CHER	or	ZHER	A is a complex Hermitian matrix

Refer to "F_SSYR/F_DSYR/F_CSYR/F_ZSYR" on page 302, and "F_CHER/F_ZHER" on page 263 for equivalent BLAS Standard subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1   uplo
INTEGER*4     n, lda, incx
REAL*4        alpha, a(lda, n), x(lenx)
CALL SSYR(uplo, n, alpha, x, incx, a, lda)
```

```
CHARACTER*1   uplo
INTEGER*4     n, lda, incx
REAL*8        alpha, a(lda, n), x(lenx)
CALL DSYR(uplo, n, alpha, x, incx, a, lda)
```

```
CHARACTER*1   uplo
INTEGER*4     n, lda, incx
REAL*4        alpha
COMPLEX*8     a(lda, n), x(lenx)
CALL CHER(uplo, n, alpha, x, incx, a, lda)
```

```
CHARACTER*1    uplo
INTEGER*4      n, lda, incx
REAL*8         alpha
COMPLEX*16     a(lda, n), x(lenx)
CALL ZHER(uplo, n, alpha, x, incx, a, lda)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	Reference and update only the lower triangle of A.
'U' or 'u'	Reference and update only the upper triangle of A.

n Number of rows and columns in matrix A and elements of vector x, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **x**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **a** or **x**.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the n-vector x.

incx Increment for the array **x**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$	x is stored forward in array **x**; that is, x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$	x is stored backward in array **x**; that is, x_i is stored in $\mathbf{x}((i-\mathbf{n})\times\mathbf{incx}+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of an n-by-n real symmetric or complex Hermitian matrix A. The other triangle of **a** is not referenced.

lda The leading dimension of array **a** as declared in the calling program unit, with $\mathbf{lda} \geq \max(\mathbf{n},1)$.

Output

a The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **a** is unchanged.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
lda < max(**n**,1)
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1

Apply a REAL*4 symmetric rank-1 update xx^T to A, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in the upper triangle of an array A whose dimensions are 10-by-10, and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,LDA,INCX
REAL*4      ALPHA,A(10,10),X(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
CALL SSYR (UPLO,N,ALPHA,X,INCX,A,LDA)
```

Example 2

Apply a COMPLEX*8 Hermitian rank-1 update $-2xx^*$ to A, where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10-by-10, and x is a complex vector 9 elements long stored in an array X of dimension 10.

```
INTEGER*4 N,LDA
COMPLEX*8 A(10,10),X(10)
N = 9
LDA = 10
CALL CHER ('LOWER',N,-2.0,X,1,A,LDA)
```

Name

SSYR2/DSYR2/CHER2/ZHER2
Rank-2 update

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-2 update

$$A \leftarrow \alpha xy^* + \bar{\alpha} yx^* + A,$$

where A is an n-by-n real symmetric or complex Hermitian matrix, α is a complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are real or complex n-vectors, and x^* and y^* are the conjugate transposes of x and y, respectively. (The conjugate of a real scalar is just the scalar, and the conjugate transpose of a real vector is simply the transpose.)

The structure of A is indicated by the name of the subprogram used:

SSYR2	or	DSYR2	A is a real symmetric matrix
CHER2	or	ZHER2	A is a complex Hermitian matrix

Refer to "F_SSYR2/F_DSYR2/F_CSYR2/F_ZSYR2" on page 304, and "F_CHER2/F_ZHER2" on page 265 for equivalent BLAS Standard subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1    uplo
INTEGER*4      n, lda, incx, incy
REAL*4         alpha, a(lda, n), x(lenx), y(leny)
CALL SSYR2(uplo, n, alpha, x, incx, y, incy, a, lda)

CHARACTER*1    uplo
INTEGER*4      n, lda, incx, incy
REAL*8         alpha, a(lda, n), x(lenx), y(leny)
CALL DSYR2(uplo, n, alpha, x, incx, y, incy, a, lda)

CHARACTER*1    uplo
INTEGER*4      n, lda, incx, incy
COMPLEX*8      alpha, a(lda, n), x(lenx), y(leny)
CALL CHER2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

```
CHARACTER*1   uplo
INTEGER*4     n, lda, incx, incy
COMPLEX*16    alpha, a(lda, n), x(lenx), y(leny)
CALL ZHER2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

Input

uplo Upper/lower triangular option for A:

- 'L' or 'l' — Reference and update only the lower triangle of A.
- 'U' or 'u' — Reference and update only the upper triangle of A.

n Number of rows and columns in matrix A and elements of vectors x and y, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **a**, **x**, or **y**.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

incx Increment for the array **x**, **incx** $\neq 0$:

- **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

y Array of length **leny** = (**n**−1)×|**incy**|+1 containing the n-vector y.

incy Increment for the array **y**, **incy** $\neq 0$:

- **incy** > 0 — y is stored forward in array **y**; that is, y_i is stored in **y**((*i*−1)×**incy**+1).
- **incy** < 0 — y is stored backward in array **y**; that is, y_i is stored in **y**((*i*−**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if y_i is stored in **y**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

	a	Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of an n-by-n real symmetric or complex Hermitian matrix A. The other triangle of **a** is not referenced.
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).
Output	**a**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **a** is unchanged.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u'
n < 0
lda < max(**n**,1)
incx = 0
incy = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'.

Example 1

Apply a REAL*4 symmetric rank-2 update xy^T+x^Ty to A, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in the upper triangle of an array A whose dimensions are 10-by-10, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,LDA,INCX,INCY
REAL*4      ALPHA,A(10,10),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
INCY = 1
CALL SSYR2 (UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $\alpha xy^*+\bar{\alpha}yx^*$ to A, where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10-by-10, α is a complex scalar, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y of dimension 10.

```
INTEGER*4 N,LDA
COMPLEX*8 ALPHA,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHER2 ('LOWER',N,ALPHA,X,1,Y,1,A,LDA)
```

Name SSYR2K/DSYR2K/CHER2K/CSYR2K/ZHER2K/ZSYR2K
Rank-2k update

Purpose These subprograms apply a symmetric or Hermitian rank-2k update to a real symmetric, complex symmetric, or complex Hermitian matrix; specifically they compute the following operations:

for symmetric C: $C \leftarrow \alpha AB^T + \bar{\alpha}AB^T + \beta C$ and $C \leftarrow \alpha A^T B + \bar{\alpha}B^T A + \beta C$

for Hermitian C: $C \leftarrow \alpha AB^* + \bar{\alpha}BA^* + \beta C$ and $C \leftarrow \alpha A^* B + \bar{\alpha}B^* A + \beta C$

where α and β are scalars, $\bar{\alpha}$ is the complex conjugate of α, C is an n-by-n real symmetric, complex symmetric, or complex Hermitian matrix, and A and B are matrices whose size, either n-by-k or k-by-n, depends on which form of the update is requested. Here, A^T and B^T are the transposes and $A^*$and B^* are the conjugate transposes of A and B, respectively. (The conjugate of a real scalar is just the scalar, and the conjugate transpose of a real matrix is simply the transpose.)

The structure of C is indicated by the name of the subprogram used:

SSYR2K	or	DSYR2K	C is a real symmetric matrix
CHER2K	or	ZHER2K	C is a complex Hermitian matrix
CSYR2K	or	ZSYR2K	C is a complex symmetric matrix

Matrix Storage Because either triangle of C can be obtained from the other, these subprograms reference and apply the update to only one triangle of C. You can supply either the upper or the lower triangle of C, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
REAL*4         alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL SSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
REAL*8         alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
REAL*4         beta
COMPLEX*8      alpha, a(lda, *), b(ldb, *), c(ldc, n)
CALL CHER2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
COMPLEX*8      alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
REAL*8         beta
COMPLEX*16     alpha, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZHER2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldb, ldc
COMPLEX*16     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Input

uplo Upper/lower triangular storage option for C:

'L' or 'l'	Reference and update only the lower triangle of C
'U' or 'u'	Reference and update only the upper triangle of C

trans Specifies the operation to be performed:

'N' or 'n'	Compute $C \leftarrow \alpha AB^T + \bar{\alpha}BA^T + \beta C$
'T' or 't'	Compute $C \leftarrow \alpha A^T B + \bar{\alpha}B^T A + \beta C$
'C' or 'c'	Compute $C \leftarrow \alpha A^* B + \bar{\alpha}B^* A + \beta C$

'T' and 't' are invalid in subprograms CHER2K and ZHER2K, and 'C' and 'c' are invalid in subprograms CSYR2K and ZSYR2K. In subprograms SSYR2K and DSYR2K, 'C' and 'c' have the same meaning as 'T' and 't'.

n Number of rows and columns in matrix C, $\mathbf{n} \geq 0$. If **n** = 0, the subprograms do not reference **a**, **b**, or **c**.

	k	Number of rows or columns in matrices A and B, depending on **trans**; refer to the description of **a** for details. **k** ≥ 0; if **k** = 0, the subprograms do not reference **a** or **b**.
	alpha	The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.
	a	Array containing the matrix A, whose size is indicated by **trans**: 'N' or 'n' A is n-by-k otherwise A is k-by-n
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (the number of rows of A,1).
	b	Array containing matrix B, which is the same size as matrix A. Refer to the description of **a** for details.
	ldb	The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max (the number of rows of B,1).
	beta	The scalar β.
	c	Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the n-by-n symmetric or Hermitian matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**n**,1).
Output	**c**	The upper or lower triangle of the updated matrix C, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **c** is unchanged.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure.

Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
n < 0
k < 0
lda too small
ldb too small
ldc < max(**m**,1)

Also, note that some of the values of **trans** listed above are invalid in subprograms CHER2K, CSYR2K, ZHER2K, and ZSYR2K.

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Apply a REAL*4 rank-6 update $AB^T + BA^T$ to an 8-by-8 real symmetric matrix C whose upper triangle is stored in the upper triangle of an array C of dimension 10-by-10, where A is an 8-by-3 real matrix stored in an array A, also of dimension 10-by-10.

```
CHARACTER*1 UPLO,TRANS
INTEGER*4   N,K,LDA,LDB,LDC
REAL*4      ALPHA,BETA,A(10,10),B(10,10),C(10,10)
UPLO = 'U'
TRANS = 'N'
N = 8
K = 3
ALPHA = 1.0
BETA = 1.0
LDA = 10
LDB = 10
LDC = 10
CALL SSYR2K (UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-4 update $-2AB^*-2BA^*$ to a 9-by-9 complex Hermitian matrix C whose lower triangle is stored in the lower triangle of an array C of dimension 10-by-10, where A is a 9-by-2 complex matrix stored in an array A of dimension 10-by-10.

```
INTEGER*4 N,K,LDA,LDB,LDC
COMPLEX*8 A(10,10),B(10,10),C(10,10)
N = 9
K = 2
LDA = 10
LDB = 10
LDC = 10
CALL CHER2K ('LOWER','NONTRANS',N,K,-2.0,A,LDA,B,LDB,
&    1.0,C,LDC)
```

Name

SSYRK/DSYRK/CHERK/CSYRK/ZHERK/ZSYRK
Rank-k update

Purpose

These subprograms apply a rank-k update to a real symmetric, complex symmetric, or complex Hermitian matrix; specifically they compute

$$C \leftarrow \alpha A A^T + \beta C, \qquad C \leftarrow \alpha A^T A + \beta C,$$
$$C \leftarrow \alpha A A^* + \beta C, \qquad C \leftarrow \alpha A^* A + \beta C,$$

where α and β are scalars, C is an n-by-n real symmetric, complex symmetric, or complex Hermitian matrix, and A is a matrix whose size, either n-by-k or k-by-n, depends on which form of the update is requested. Here, A^T and A^* are the transpose and conjugate transpose of A, respectively.

The structure of C is indicated by the name of the subprogram used:

SSYRK	or	DSYRK	C is a real symmetric matrix
CHERK	or	ZHERK	C is a complex Hermitian matrix
CSYRK	or	ZSYRK	C is a complex symmetric matrix

Matrix Storage

Because either triangle of C can be obtained from the other, these subprograms reference and apply the update to only one triangle of C. You can supply either the upper or the lower triangle of C, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1     uplo, trans
INTEGER*4       n, k, lda, ldc
REAL*4          alpha, beta, a(lda, *), c(ldc, n)
CALL SSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

CHARACTER*1     uplo, trans
INTEGER*4       n, k, lda, ldc
REAL*8          alpha, beta, a(lda, *), c(ldc, n)
CALL DSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

CHARACTER*1     uplo, trans
INTEGER*4       n, k, lda, ldc
REAL*4          alpha, beta
COMPLEX*8       a(lda, *), c(ldc, n)
CALL CHERK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
```

```
CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldc
COMPLEX*8      alpha, beta, a(lda, *), c(ldc, n)
CALL CSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldc
REAL*8         alpha, beta
COMPLEX*16     a(lda, *), c(ldc, n)
CALL ZHERK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

CHARACTER*1    uplo, trans
INTEGER*4      n, k, lda, ldc
COMPLEX*16     alpha, beta, a(lda, *), c(ldc, n)
CALL ZSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
```

Input

uplo Upper/lower triangular storage option for C:

'L' or 'l'	Reference and update only the lower triangle of C
'U' or 'u'	Reference and update only the upper triangle of C

trans Specifies the operation to be performed:

'N' or 'n'	Compute $C \leftarrow \alpha AA^T + \beta C$
'T' or 't'	Compute $C \leftarrow \alpha A^T A + \beta C$
'C' or 'c'	Compute $C \leftarrow \alpha A^* A + \beta C$

'T' and 't' are invalid in subprograms CHERK and ZHERK, and 'C' and 'c' are invalid in subprograms CSYRK and ZSYRK. In subprograms SSYRK and DSYRK, 'C' and 'c' have the same meaning as 'T' and 't'.

n Number of rows and columns in matrix C, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **c**.

k Number of rows or columns in matrix A, $\mathbf{k} \geq 0$, depending on **trans**; refer to description of **A** for details. If $\mathbf{k} = 0$, the subprograms do not reference **a**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a**.

a Array containing the matrix A, whose size is indicated by **trans**:

'N' or 'n'	A is n-by-k
otherwise	A is k-by-n

	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (the number of rows of A,1).
	beta	The scalar β.
	c	Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the n-by-n symmetric or Hermitian matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**n**,1).
Output	**c**	The upper or lower triangle of the updated C matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **c** is unchanged.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
n < 0
k < 0
lda too small
ldc < max(**m**,1)

Also, some values of **trans** listed above are invalid in subprograms CHERK, CSYRK, ZHERK, and ZSYRK.

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Apply a REAL*4 rank-6 update AA^T to an 8-by-8 real symmetric matrix C whose upper triangle is stored in the upper triangle of an array C of dimension 10-by-10, where A is an 8-by-6 real matrix stored in an array A, also of dimension 10-by-10.

```
CHARACTER*1 UPLO,TRANS
INTEGER*4   N,K,LDA,LDC
REAL*4      ALPHA,BETA,A(10,10),C(10,10)
UPLO = 'U'
TRANS = 'N'
N = 8
K = 6
ALPHA = 1.0
BETA = 1.0
LDA = 10
LDC = 10
CALL SSYRK (UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $-2AA^*$ to a 9-by-9 complex Hermitian matrix C whose lower triangle is stored in the lower triangle of an array C of dimension 10-by-10, where A is a 9-by-2 complex matrix stored in an array A of dimension 10-by-10.

```
INTEGER*4 N,K,LDA,LDC
COMPLEX*8 A(10,10),C(10,10)
N = 9
K = 2
LDA = 10
LDC = 10
CALL CHERK ('LOWER','NONTRANS',N,K,-2.0,A,LDA,1.0,C,LDC)
```

Name

STBMV/DTBMV/CTBMV/ZTBMV
Matrix-vector multiply

Purpose

Given an n-by-n upper- or lower-triangular band matrix A and an n-vector x, these subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A^T is the transpose of A, and A^* is the conjugate transpose. Specifically, these subprograms compute matrix-vector products of the forms

$$x \leftarrow Ax, x \leftarrow A^Tx, \text{ and } x \leftarrow A^*x.$$

A lower-triangular band matrix is a matrix whose strict upper triangle is zero and whose nonzero lower-triangular elements all are on or fairly near the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $0 \leq i-j \leq kd$ for some integer kd.

In contrast, an upper-triangular band matrix is a matrix whose strict lower triangle is zero and whose nonzero upper-triangular elements all are on or fairly near the principal diagonal, that is, with $a_{ij} \neq 0$ only if $0 \leq j-i \leq kd$.

Refer to "F_STBMV/F_DTBMV/F_CTBMV/F_ZTBMV" on page 306 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that can be nonzero. The following examples illustrate the storage of triangular band matrices.

Lower triangular storage.

If A is a 9-by-9 lower-triangular band matrix with bandwidth $kd = 3$, for example,

11	0	0	0	0	0	0	0	0
21	22	0	0	0	0	0	0	0
31	32	33	0	0	0	0	0	0
41	42	43	44	0	0	0	0	0
0	52	53	54	55	0	0	0	0
0	0	63	64	65	66	0	0	0
0	0	0	74	75	76	77	0	0
0	0	0	0	85	86	87	88	0
0	0	0	0	0	96	97	98	99

the lower triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*
41	52	63	74	85	96	*	*	*

where asterisks represent elements in the kd-by-kd triangle at the lower-right corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**$(1+i-j,j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in the first row, the first subdiagonal in the second row, and so on.

Upper triangular storage.

If A is a 9-by-9 upper-triangular band matrix with bandwidth $kd = 3$, for example,

11	12	13	14	0	0	0	0	0
0	22	23	24	25	0	0	0	0
0	0	33	34	35	36	0	0	0
0	0	0	44	45	46	47	0	0
0	0	0	0	55	56	57	58	0
0	0	0	0	0	66	67	68	69
0	0	0	0	0	0	77	78	79
0	0	0	0	0	0	0	88	89
0	0	0	0	0	0	0	0	99

the upper triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99

where asterisks represent elements in the kd-by-kd triangle at the upper-left corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**($kd+1+i-j,j$). Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in row $kd+1$, the first superdiagonal starting in the second position in row kd, and so on.

Usage

```
CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
REAL*4         ab(ldab, n), x(lenx)
CALL STBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
REAL*8         ab(ldab, n), x(lenx)
CALL DTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
COMPLEX*8      ab(ldab, n), x(lenx)
CALL CTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
COMPLEX*16     ab(ldab, n), x(lenx)
CALL ZTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	A is lower triangular
'U' or 'u'	A is upper triangular

trans Transposition option for A:

'N' or 'n'	Compute $x \leftarrow Ax$
'T' or 't'	Compute $x \leftarrow A^T x$
'C' or 'c'	Compute $x \leftarrow A^* x$

where A^T is the transpose of A, and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', diagonal elements of A are not referenced, but space must be reserved for them.

	n	Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ab** or **x**.
	kd	The number of nonzero diagonals above or below the principal diagonal. If **uplo** is supplied as 'U' or 'u', **kd** specifies the number of nonzero diagonals above the principal diagonal. If **uplo** is supplied as 'L' or 'l', **kd** specifies the number of nonzero diagonals below the principal diagonal.
	ab	Array containing the n-by-n triangular band matrix A in the compressed form described above. The columns of the band of A are stored in the columns of **ab**, and the diagonals of the band of A are stored in the rows of **ab**.
	ldab	The leading dimension of array **ab** as declared in the calling program unit, with $\mathbf{ldab} \geq \mathbf{kd}+1$.
	x	Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times\lvert\mathbf{incx}\rvert+1$ containing the input vector x.
	incx	Increment for the array **x**, $\mathbf{incx} \neq 0$:
		incx > 0 — x is stored forward in array **x**; that is, x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
		incx < 0 — x is stored backward in array **x**; that is, x_i is stored in $\mathbf{x}((i-\mathbf{n})\times\mathbf{incx}+1)$.
		Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**x**	The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure.

Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
kd < 0
ldab < **kd**+1
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product Ax, where A is a 75-by-75 unit-diagonal, lower-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,KD,LDAB,INCX
REAL*4      AB(25,100),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
KD = 15
LDAB = 25
INCX = 1
CALL STBMV (UPLO,TRANS,DIAG,N,KD,AB,LDAB,X,INCX)
```

Example 2 Form the REAL*8 matrix-vector product A^Tx, where A is a 75-by-75 nonunit-diagonal, upper-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
INTEGER*4 N,KD,LDAB
REAL*4    AB(25,100),X(100)
N = 75
KD = 15
LDAB = 25
CALL DTBMV ('UPPER','TRANSPOSE','NONUNIT',N,KD,AB,LDAB,X,1)
```

Name

STBSV/DTBSV/CTBSV/ZTBSV
Solve triangular band system

Purpose

Given an n-by-n upper- or lower-triangular band matrix A and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This operation is the forward elimination or back substitution step of Gaussian elimination for band matrices. Optionally, A can be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A.

A lower-triangular band matrix is a matrix whose strict upper triangle is zero and whose nonzero lower-triangular elements all are on, or fairly near, the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $0 \leq i-j \leq kd$ for some integer kd. In contrast, an upper-triangular band matrix is a matrix whose strict lower triangle is zero and whose nonzero upper-triangular elements all are on, or fairly near, the principal diagonal, but with $a_{ij} \neq 0$ only if $0 \leq j-i \leq kd$.

Specifically, these subprograms compute

$$x \leftarrow A^{-1}x,\ x \leftarrow A^{-T}x,\ \text{and}\ x \leftarrow A^{-*}x$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LAPACK band equation solvers. As such, they are intended to supplement the equation solvers but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LAPACK subprograms have been recoded to call these routines.

Refer to "F_STBSV/F_DTBSV/F_CTBSV/F_ZTBSV" on page 308 for details about the equivalent BLAS Standard subprograms.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that can be nonzero. The following examples illustrate the storage of triangular band matrices.

Lower triangular storage

If A is a 9-by-9 lower-triangular band matrix with bandwidth $kd = 3$, for example,

11	0	0	0	0	0	0	0	0
21	22	0	0	0	0	0	0	0
31	32	33	0	0	0	0	0	0
41	42	43	44	0	0	0	0	0
0	52	53	54	55	0	0	0	0
0	0	63	64	65	66	0	0	0
0	0	0	74	75	76	77	0	0
0	0	0	0	85	86	87	88	0
0	0	0	0	0	96	97	98	99

the lower triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*
41	52	63	74	85	96	*	*	*

where asterisks represent elements in the kd-by-kd triangle at the lower-right corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**$(1+i-j,j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in the first row, the first subdiagonal in the second row, and so on.

Upper triangular storage

If A is a 9-by-9 upper-triangular band matrix with bandwidth $kd = 3$, for example,

11	12	13	14	0	0	0	0	0
0	22	23	24	25	0	0	0	0
0	0	33	34	35	36	0	0	0
0	0	0	44	45	46	47	0	0
0	0	0	0	55	56	57	58	0
0	0	0	0	0	66	67	68	69
0	0	0	0	0	0	77	78	79
0	0	0	0	0	0	0	88	89
0	0	0	0	0	0	0	0	99

the upper triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99

where asterisks represent elements in the kd-by-kd triangle at the upper-left corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**($kd+1+i-j,j$). Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in row $kd+1$, the first superdiagonal starting in the second position in row kd, and so on.

Usage

```
CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
REAL*4         ab(ldab, n), x(lenx)
CALL STBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
REAL*8         ab(ldab, n), x(lenx)
CALL DTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
COMPLEX*8      ab(ldab, n), x(lenx)
CALL CTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)
```

```
CHARACTER*1    uplo, trans, diag
INTEGER*4      n, kd, ldab, incx
COMPLEX*16     ab(ldab, n), x(lenx)
CALL ZTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	Solve lower-triangular band system (forward elimination)
'U' or 'u'	Solve upper-triangular band system (back substitution)

trans Transposition option for A:

'N' or 'n'	Compute $x \leftarrow A^{-1}x$
'T' or 't'	Compute $x \leftarrow A^{-T}x$
'C' or 'c'	Compute $x \leftarrow A^{-*}x$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', diagonal elements of A are not referenced, but space must be reserved for them.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ab** or **x**.

kd The number of nonzero diagonals above or below the principal diagonal. If **uplo** is supplied as 'U' or 'u', **kd** specifies the number of nonzero diagonals above the principal diagonal. If **uplo** is supplied as 'L' or 'l', **kd** specifies the number of nonzero diagonals below the principal diagonal.

	ab	Array containing the n-by-n triangular band matrix A in the compressed form described above. The columns of the band of A are stored in the columns of **ab**, and the diagonals of the band of A are stored in the rows of **ab**.
	ldab	The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **kd**+1.
	x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the right-hand-side n-vector x.
	incx	Increment for the array **x**, **incx** ≠ 0: **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((i−1)×**incx**+1). **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((i−**n**)×**incx**+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**x**	The solution vector of the triangular band system replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition causes a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure.

Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
kd < 0
ldab < **kd**+1
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*4 forward elimination using the 75-by-75 unit-diagonal lower-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,KD,LDAB,INCX
REAL*4      AB(25,100),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
KD = 15
LDAB = 25
INCX = 1
CALL STBSV (UPLO,TRANS,DIAG,N,KD,AB,LDAB,X,INCX)
```

Example 2 Perform REAL*4 back substitution using the 75-by-75 nonunit-diagonal, upper-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
INTEGER*4 N,KD,LDAB
REAL*4    AB(25,100),X(100)
N = 75
KD = 15
LDAB = 25
CALL STBSV ('UPPER','NONTRANS','NONUNIT',N,KD,AB,LDAB,X,1)
```

Name STPMV/DTPMV/CTPMV/ZTPMV
Matrix-vector multiply

Purpose Given an n-by-n upper- or lower-triangular matrix A stored in packed form as described in "Matrix Storage" and an n-vector x, these subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A^T is the transpose of A, and A^* is the conjugate transpose of A. Specifically, these subprograms compute matrix-vector products of the forms

$$x \leftarrow Ax, x \leftarrow A^Tx, \text{ and } x \leftarrow A^*x.$$

Refer to "F_STPMV/F_DTPMV/F_CTPMV/F_ZTPMV" on page 310 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage You supply the upper or lower triangle of A, stored column-by-column in packed form in a 1-dimensional array. This saves memory compared to storing the entire matrix.

The following examples illustrate the packed storage of a triangular matrix.

Upper triangular matrix

If A is

$$\begin{matrix} 11 & 12 & 13 & 14 \\ 0 & 22 & 23 & 24 \\ 0 & 0 & 33 & 34 \\ 0 & 0 & 0 & 44 \end{matrix}$$

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper-triangular matrix element a_{ij} is stored in array element **ap**(i+(j×(j–1))/2).

Lower triangular matrix

If A is

$$\begin{matrix} 11 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 \\ 31 & 32 & 33 & 0 \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower-triangular matrix element a_{ij} is stored in array element **ap**(i+((j–1)×(2n–j))/2).

Usage

```
CHARACTER*1     uplo, trans, diag
INTEGER*4       n, incx
REAL*4          ap(lenap), x(lenx)
CALL STPMV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*4       n, incx
REAL*8          ap(lenap), x(lenx)
CALL DTPMV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*4       n, incx
COMPLEX*8       ap(lenap), x(lenx)
CALL CTPMV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*4       n, incx
COMPLEX*16      ap(lenap), x(lenx)
CALL ZTPMV(uplo, trans, diag, n, ap, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	A is lower triangular
'U' or 'u'	A is upper triangular

trans Transposition option for A:

'N' or 'n'	Compute $x \leftarrow Ax$
'T' or 't'	Compute $x \leftarrow A^T x$
'C' or 'c'	Compute $x \leftarrow A^* x$

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

	diag	Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not: 'N' or 'n' — The diagonal of A is stored in the array 'U' or 'u' — The diagonal of A consists of unstored ones When **diag** is supplied as 'U' or 'u', the diagonal elements are not referenced.
	n	Number of rows and columns in matrix A, **n** ≥ 0. If **n** = 0, the subprograms do not reference **ap** or **x**.
	ap	Array of length **lenap** = **n**×(**n**+1)/2 containing the n-by-n triangular matrix A, stored by columns in the packed form described above. Space must be left for the diagonal elements of A even when **diag** is supplied as 'U' or 'u'.
	x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the input vector x.
	incx	Increment for the array **x**, **incx** ≠ 0: **incx** > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1). **incx** < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**x**	The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product Ax, where A is a 9-by-9 unit-diagonal, lower-triangular real matrix stored in packed form in an array AP of dimension 55 and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,INCX
REAL*4      AP(55),X(10)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 9
INCX = 1
CALL STPMV (UPLO,TRANS,DIAG,N,AP,X,INCX)
```

Example 2 Form the REAL*8 matrix-vector product A^Tx, where A is a 9-by-9 nonunit-diagonal, upper-triangular real matrix stored in packed form in an array AP of dimension 55 and x is a real vector 9 elements long stored in an array X of dimension 10.

```
INTEGER*4 N
REAL*8    AP(55),X(10)
N = 6
CALL DTPMV ('UPPER','TRANSPOSE','NONUNIT',N,AP,X,1)
```

Name

STPSV/DTPSV/CTPSV/ZTPSV
Solve triangular system

Purpose

Given an n-by-n upper- or lower-triangular matrix A stored in packed form as described in "Matrix Storage" and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A can be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, \text{ and } x \leftarrow A^{-*}x,$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LAPACK linear equation solvers. As such, they are intended to supplement but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LAPACK subprograms have been recoded to call these subprograms.

Refer to "F_STPSV/F_DTPSV/F_CTPSV/F_ZTPSV" on page 312 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

You supply the upper or lower triangle of A, stored column-by-column in packed form in a 1-dimensional array. This saves memory compared to storing the entire matrix.

The following examples illustrate the packed storage of a triangular matrix.

Upper triangular matrix

If A is

$$\begin{matrix} 11 & 12 & 13 & 14 \\ 0 & 22 & 23 & 24 \\ 0 & 0 & 33 & 34 \\ 0 & 0 & 0 & 44 \end{matrix}$$

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper-triangular matrix element a_{ij} is stored in array element **ap**(i+(j×(j–1))/2).

Lower triangular matrix

If A is

$$\begin{matrix} 11 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 \\ 31 & 32 & 33 & 0 \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower-triangular matrix element a_{ij} is stored in array element **ap**($i+((j-1)\times(2n-j))/2$).

Usage

```
CHARACTER*1    uplo, trans, diag
INTEGER*4      n, incx
REAL*4         ap(lenap), x(lenx)
CALL STPSV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, incx
REAL*8         ap(lenap), x(lenx)
CALL DTPSV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, incx
COMPLEX*8      ap(lenap), x(lenx)
CALL CTPSV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, incx
COMPLEX*16     ap(lenap), x(lenx)
CALL ZTPSV(uplo, trans, diag, n, ap, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	Solve lower-triangular system (forward elimination)
'U' or 'u'	Solve upper-triangular system (back substitution)

trans Transposition option for A:

'N' or 'n'	Compute $x \leftarrow A^{-1}x$
'T' or 't'	Compute $x \leftarrow A^{-T}x$
'C' or 'c'	Compute $x \leftarrow A^{-*}x$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements are not referenced.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ap** or **x**.

ap Array of length **lenap** = n×(n+1)/2 containing the n-by-n triangular matrix A, stored by columns in the packed form described above. Space must be left for the diagonal elements of A even when **diag** is supplied as 'U' or 'u'.

x Array of length **lenx** = (n−1)×|**incx**|+1 containing the right-hand-side n-vector x.

incx Increment for the array **x**, **incx** ≠ 0:

incx > 0	x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1)
incx < 0	x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1)

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x The solution vector of the triangular system replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition causes a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*4 forward elimination using a 75-by-75 unit-diagonal, lower-triangular real matrix stored in packed form in an array AP of dimension 5500, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,INCX
REAL*4      AP(5500),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
INCX = 1
CALL STPSV (UPLO,TRANS,DIAG,N,AP,X,INCX)
```

Example 2 Perform REAL*4 back substitution using a 75-by-75 nonunit-diagonal, upper-triangular real matrix stored in packed form in an array AP of dimension 5500, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
INTEGER*4 N
REAL*4    AP(5500),X(100)
N = 75
CALL STPSV ('UPPER','NONTRANS','NONUNIT',N,AP,X,1)
```

Name STRMM/DTRMM/CTRMM/ZTRMM
Triangular matrix-matrix multiply

Purpose Given a scalar α, an m-by-n matrix B, and an upper- or lower-triangular matrix A, these subprograms compute either of the matrix-matrix products αAB or αBA. The size of A, either m-by-m or n-by-n, depends on which matrix product is requested. Optionally, A can be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. The resulting matrix product overwrites the input B matrix. Specifically, these subprograms compute matrix products of the forms

$$B \leftarrow \alpha AB, \quad B \leftarrow \alpha A^T B, \quad B \leftarrow \alpha A^* B,$$
$$B \leftarrow \alpha BA, \quad B \leftarrow \alpha BA^T, \quad B \leftarrow \alpha BA^*.$$

Matrix Storage For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also are not referenced.

Usage

```
CHARACTER*1    side, uplo, transa, diag
INTEGER*4      m, n, lda, ldb
REAL*4         alpha, a(lda, *), b(ldb, *)
CALL STRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1    side, uplo, transa, diag
INTEGER*4      m, n, lda, ldb
REAL*8         alpha, a(lda, *), b(ldb, *)
CALL DTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1    side, uplo, transa, diag
INTEGER*4      m, n, lda, ldb
COMPLEX*8      alpha, a(lda, *), b(ldb, *)
CALL CTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1    side, uplo, transa, diag
INTEGER*4      m, n, lda, ldb
COMPLEX*16     alpha, a(lda, *), b(ldb, *)
CALL ZTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

Input

side Specifies whether triangular matrix A is the left or right matrix operand:

'L' or 'l'	A is the left matrix operand: for example, $B \leftarrow \alpha AB$
'R' or 'r'	A is the right matrix operand: for example, $B \leftarrow \alpha BA$

uplo Upper/lower triangular option for A:

'L' or 'l'	A is a lower-triangular matrix
'U' or 'u'	A is an upper-triangular matrix

transa Transposition option for A:

'N' or 'n'	Use matrix A directly
'T' or 't'	Use A^T, the transpose of A
'C' or 'c'	Use A*, the conjugate transpose of A

In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the A matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements of A are not referenced.

m Number of rows in matrix B, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $B \leftarrow 0$ without referencing **a**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper- or lower-triangular matrix A, whose size is indicated by **side**:

'L' or 'l'	A is m-by-m
'R' or 'r'	A is n-by-n

The other triangle of **a** is not referenced. Not used as input if **alpha** = 0.

	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (the number of rows of A,1).
	b	Array containing the m-by-n matrix B. Not used as input if **alpha** = 0.
	ldb	The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max(**m**,1).
Output	**b**	The indicated matrix product replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

side ≠ 'L' or 'l' or 'R' or 'r'
uplo ≠ 'L' or 'l' or 'U' or 'u'
transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
m < 0
n < 0
lda too small
ldb < max(**m**,1)

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **transa** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix product AB, where A is a 6-by-6 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 10-by-10 and B is a 6-by-8 real matrix stored in an array B of dimension 10-by-10. The matrix product overwrites the input B matrix.

```
CHARACTER*1 SIDE,UPLO,TRANSA,DIAG
INTEGER*4   M,N,LDA,LDB
REAL*4      ALPHA,A(10,10),B(10,10)
SIDE = 'L'
UPLO = 'U'
TRANSA = 'N'
DIAG = 'N'
M = 6
N = 8
ALPHA = 1.0
LDA = 10
LDB = 10
CALL STRMM (SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB)
```

Example 2 Form the REAL*8 matrix product qBA^T, where q is a real scalar, B is a 6-by-8 real matrix stored in an array B of dimension 10-by-10, and A is a 8-by-8 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10-by-10. The matrix product overwrites the input B matrix.

```
INTEGER*4 M,N,LDA,LDB
REAL*8    Q,A(10,10),B(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
CALL DTRMM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)
```

Name

STRMV/DTRMV/CTRMV/ZTRMV
Matrix-vector multiply

Purpose

Given an n-by-n upper- or lower-triangular matrix A and an n-vector x, these subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A^T is the transpose of A, and A^* is the conjugate transpose of A. Specifically, these subprograms compute matrix-vector products of the forms

$$x \leftarrow Ax, x \leftarrow A^Tx, \text{ and } x \leftarrow A^*x.$$

Refer to "F_STRMV/F_DTRMV/F_CTRMV/F_ZTRMV" on page 314 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also is not referenced.

Usage

```
CHARACTER*1    uplo, trans, diag
INTEGER*4      n, lda, incx
REAL*4         a(lda, n), x(lenx)
CALL STRMV(uplo, trans, diag, n, a, lda, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, lda, incx
REAL*8         a(lda, n), x(lenx)
CALL DTRMV(uplo, trans, diag, n, a, lda, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, lda, incx
COMPLEX*8      a(lda, n), x(lenx)
CALL CTRMV(uplo, trans, diag, n, a, lda, x, incx)

CHARACTER*1    uplo, trans, diag
INTEGER*4      n, lda, incx
COMPLEX*16     a(lda, n), x(lenx)
CALL ZTRMV(uplo, trans, diag, n, a, lda, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	A is lower triangular
'U' or 'u'	A is upper triangular

The other triangle of the array **a** is not referenced.

	trans	Transposition option for A:
		'N' or 'n' — Compute $x \leftarrow Ax$
		'T' or 't' — Compute $x \leftarrow A^T x$
		'C' or 'c' — Compute $x \leftarrow A^* x$
		where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.
	diag	Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not:
		'N' or 'n' — The diagonal of A is stored in the array
		'U' or 'u' — The diagonal of A consists of unstored ones
		When **diag** is supplied as 'U' or 'u', the diagonal elements are not referenced.
	n	Number of rows and columns in matrix A, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **x**.
	a	Array containing the n-by-n triangular matrix A.
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).
	x	Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the input vector x.
	incx	Increment for the array **x**, **incx** ≠ 0:
		incx > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
		incx < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).
		Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**x**	The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
lda < max(**n**,1)
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement can be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product Ax, where A is a 9-by-9 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10-by-10, and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,LDA,INCX
REAL*4      A(10,10),X(10)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 9
LDA = 10
INCX = 1
CALL STRMV (UPLO,TRANS,DIAG,N,A,LDA,X,INCX)
```

Example 2 Form the REAL*8 matrix-vector product A^Tx, where A is a 9-by-9 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 10-by-10, and x is a real vector 9 elements long stored in an array X of dimension 10.

```
INTEGER*4 N,LDA
REAL*8    A(10,10),X(10)
N = 6
LDA = 10
CALL DTRMV ('UPPER','TRANSPOSE','NONUNIT',N,A,LDA,X,1)
```

Name

STRSM/DTRSM/CTRSM/ZTRSM
Solve triangular systems

Purpose

Given a scalar α, an upper- or lower-triangular matrix A and an m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$ or αBA^{-1}. The size of A, either m-by-m or n-by-n, depends on which matrix solution is requested. Optionally, A^{-1} can be replaced by A^{-T}, the inverse of the transpose of A, or by A^{-*}, the inverse of the conjugate transpose of A. The resulting matrix solution overwrites the input B matrix. Specifically, these subprograms compute matrix solutions of the forms

$$B \leftarrow \alpha A^{-1}B, \quad B \leftarrow \alpha A^{-T}B, \quad B \leftarrow \alpha A^{-*}B,$$

$$B \leftarrow \alpha BA^{-1}, \quad B \leftarrow \alpha BA^{-T}, \quad B \leftarrow \alpha BA^{-*}.$$

Refer to "F_STRSM/F_DTRSM/F_CTRSM/F_ZTRSM" on page 318 for a description of the equivalent BLAS Standard subprograms.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also is not referenced.

Usage

```
CHARACTER*1     side, uplo, transa, diag
INTEGER*4       m, n, lda, ldb
REAL*4          alpha, a(lda, *), b(ldb, *)
CALL STRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1     side, uplo, transa, diag
INTEGER*4       m, n, lda, ldb
REAL*8          alpha, a(lda, *), b(ldb, *)
CALL DTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1     side, uplo, transa, diag
INTEGER*4       m, n, lda, ldb
COMPLEX*8       alpha, a(lda, *), b(ldb, *)
CALL CTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1     side, uplo, transa, diag
INTEGER*4       m, n, lda, ldb
COMPLEX*16      alpha, a(lda, *), b(ldb, *)
CALL ZTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

Input

side Specifies whether triangular matrix A is the left or right matrix operand:

'L' or 'l'	A is the left matrix operand: for example, $B \leftarrow \alpha A^{-1}B$
'R' or 'r'	A is the right matrix operand: for example, $B \leftarrow \alpha BA^{-1}$

uplo Upper/lower triangular option for A:

'L' or 'l'	A is a lower-triangular matrix
'U' or 'u'	A is an upper-triangular matrix

transa Transposition option for A:

'N' or 'n'	Use matrix A^{-1}
'T' or 't'	Use A^{-T}, the inverse of the transpose of A
'C' or 'c'	Use A^{-*}, the inverse of the conjugate transpose of A

In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the A matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements of A are not referenced.

m Number of rows in matrix B, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $B \leftarrow 0$ without referencing **a**.

	a	Array whose upper or lower triangle, as specified by **uplo**, contains the upper- or lower-triangular matrix A, whose size is indicated by **side**: 'L' or 'l' A is m-by-m 'R' or 'r' A is n-by-n The other triangle of **a** is not referenced. Not used as input if **alpha** = 0.
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max (the number of rows of A,1).
	b	Array containing the m-by-n matrix B. Not used as input if **alpha** = 0.
	ldb	The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max(**m**,1).
Output	**b**	The indicated matrix solution replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are:

side ≠ 'L' or 'l' or 'R' or 'r'
uplo ≠ 'L' or 'l' or 'U' or 'u'
transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
m < 0
n < 0
lda too small
ldb < max(**m**,1)

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **transa** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix solution $A^{-1}B$, where A is a 6-by-6 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 10-by-10 and B is a 6-by-8 real matrix stored in an array B of dimension 10-by-10. The matrix solution overwrites the input B matrix.

```
CHARACTER*1 SIDE,UPLO,TRANSA,DIAG
INTEGER*4   M,N,LDA,LDB
REAL*4      ALPHA,A(10,10),B(10,10)
SIDE = 'L'
UPLO = 'U'
TRANSA = 'N'
DIAG = 'N'
M = 6
N = 8
ALPHA = 1.0
LDA = 10
LDB = 10
CALL STRSM (SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB)
```

Example 2 Form the REAL*8 matrix solution qBA^{-T}, where q is a real scalar, B is a 6-by-8 real matrix stored in an array B of dimension 10-by-10, and A is a 8-by-8 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10-by-10. The matrix solution overwrites the input B matrix.

```
INTEGER*4 M,N,LDA,LDB
REAL*8    Q,A(10,10),B(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
CALL DTRSM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)
```

Name

STRSV/DTRSV/CTRSV/ZTRSV
Solve triangular system

Purpose

Given an n-by-n upper- or lower-triangular matrix A and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A can be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, \text{ and } x \leftarrow A^{-*}x,$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LAPACK linear equation solvers. As such, they are intended to supplement but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LAPACK subprograms have been recoded to call these subprograms.

Refer to "F_STRSV/F_DTRSV/F_CTRSV/F_ZTRSV" on page 321 for details of the equivalent BLAS Standard subprograms.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also are not referenced.

Usage

```
CHARACTER*1   uplo, trans, diag
INTEGER*4     n, lda, incx
REAL*4        a(lda, n), x(lenx)
CALL STRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1   uplo, trans, diag
INTEGER*4     n, lda, incx
REAL*8        a(lda, n), x(lenx)
CALL DTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1   uplo, trans, diag
INTEGER*4     n, lda, incx
COMPLEX*8     a(lda, n), x(lenx)
CALL CTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1   uplo, trans, diag
INTEGER*4     n, lda, incx
COMPLEX*16    a(lda, n), x(lenx)
CALL ZTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l'	Solve lower-triangular system (forward elimination)
'U' or 'u'	Solve upper-triangular system (back substitution)

The other triangle of the array **a** is not referenced.

trans Transposition option for A:

'N' or 'n'	Compute $x \leftarrow A^{-1}x$
'T' or 't'	Compute $x \leftarrow A^{-T}x$
'C' or 'c'	Compute $x \leftarrow A^{-*}x$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the matrix is unit triangular, that is, $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements are not referenced.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a** or **x**.

a Array containing the n-by-n triangular matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with $\mathbf{lda} \geq \max(\mathbf{n},1)$.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the right-hand-side n-vector x.

	incx	Increment for the array **x**, **incx** ≠ 0:
		incx > 0 — x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
		incx < 0 — x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).
		Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
Output	**x**	The solution vector of the triangular system replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition causes a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u'
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c'
diag ≠ 'N' or 'n' or 'U' or 'u'
n < 0
lda < max(**n**,1)
incx = 0

Actual character arguments in a subroutine call can be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*4 forward elimination using the 75-by-75 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 100-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,LDA,INCX
REAL*4      A(100,100),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
LDA = 100
INCX = 1
CALL STRSV (UPLO,TRANS,DIAG,N,A,LDA,X,INCX)
```

Example 2 Perform REAL*4 back substitution using the 75-by-75 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 100-by-100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
INTEGER*4 N,LDA
REAL*4    A(100,100),X(100)
N = 75
LDA = 100
CALL STRSV ('UPPER','NONTRANS','NONUNIT',N,A,LDA,X,1)
```

Name

XERBLA
Error handler

Purpose

This subprogram is the error handler for many of the subprograms in this chapter, as indicated in the "Notes" section in the applicable subprogram descriptions. As supplied in VECLIB, XERBLA writes the following error message onto the standard error file:

```
********************************************************************************
* XERBLA: subprogram name called with invalid value of argument number iarg *
********************************************************************************
```

where *name* is the name of the subprogram in which the error was detected, and *iarg* is the argument number of the offending argument. For example, in SGEMV, **trans** is argument number 1 and **m** is argument number 2. XERBLA then terminates execution with a nonzero exit status.

You can supply a version of XERBLA that alters this action. Be aware that other subprograms, including many in LAPACK, also call XERBLA. All BLAS, VECLIB and LAPACK subprograms that call XERBLA follow the **CALL XERBLA** statement with a **RETURN** statement, so your version of XERBLA can exit with a **RETURN** statement. However, many of those subprograms do not have a status response variable in their argument list that could be used to alert the caller. If you write an XERBLA that does not end with a **STOP** statement, you need some other mechanism to detect errors occurring in those subprograms. One such mechanism is a flag in a common block that is set by your XERBLA and tested by the calling program after calls where errors could be detected.

Usage

```
CHARACTER*6    name
INTEGER*4      iarg
CALL XERBLA(name, iarg)
```

Input

name	The name of the subprogram in which the error was detected.
iarg	The number of the argument that was found to be in error.

Notes

This subprogram conforms to specifications of the Level 2 and 3 BLAS and LAPACK.

BLAS Standard routines

Name F_CHBMV/F_ZHBMV
Hermitian banded matrix-vector multiply

Purpose F_xHBMV multiplies a vector x by a Hermitian banded matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero, or if β is equal to one and α is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

Refer to "SSBMV/DSBMV/CHBMV/ZHBMV" on page 182 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage Because it is not necessary to store or operate on the zeros outside the band of A, and because either triangle of A can be obtained from the other, you only need to provide the band within one triangle of A. Compared to storing the entire matrix, this can save memory in two ways: only the elements within the band are stored and of them only the upper or the lower triangle. Refer to "SSBMV/DSBMV/CHBMV/ZHBMV" on page 182 for an example of the storage of symmetric band matrices.

Usage

```
INTEGER      INCX, INCY, K, LDA, N, UPLO
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CHBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)
```

```
INTEGER      INCX, INCY, K, LDA, N, UPLO
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZHBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	K	The number of non zero diagonals above or below the principal diagonal.
	ALPHA	The scalar ALPHA.
	A	COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **k**+1, an error condition is generated
	X	COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	Y	COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with A=A*

Name

F_CHEMV/F_ZHEMV
Hermitian matrix-vector multiply

Purpose

F_xHEMV multiplies a vector x by a Hermitian matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if β is equal to one and α is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

Refer to "SSYMV/DSYMV/CHEMV/ZHEMV" on page 204 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The other triangle of the array is not referenced.

Usage

```
INTEGER      INCX, INCY, LDA, N, UPLO
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CHEMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)

INTEGER      INCX, INCY, LDA, N, UPLO
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZHEMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)
```

Input

UPLO	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
ALPHA	The scalar ALPHA.
A	COMPLEX array, dimension (LDA, N).
LDA	Leading dimension of array A. If **lda** < 1 or **lda** < **n**, an error condition is generated.
X	COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	Y	COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with A=A*

Name

F_CHER/F_ZHER
Hermitian rank-1 update

Purpose

F_xHER performs the Hermitian rank-1 update

$$A \leftarrow \alpha x x^* + \beta A^* \quad \text{with} \quad A=A^*$$

where A is an n-by-n complex Hermitian matrix, α is a real scalar, x is a complex n-vector, and x^* is the conjugate transpose of x. The routine returns immediately if n is less than or equal to zero.

Refer to "SSYR/DSYR/CHER/ZHER" on page 208 for a description of the HP MLIB legacy BLAS subprogram for rank-1 update.

Matrix Storage

Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
INTEGER      INCX, LDA, N, UPLO
REAL*4       ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * )
SUBROUTINE F_CHER (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)

INTEGER      INCX, LDA, N, UPLO
REAL*8       ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * )
SUBROUTINE F_ZHER (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)
```

Input

UPLO	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
N	Number of elements of vector x.
ALPHA	The scalar ALPHA.
X	COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
BETA	The scalar BETA.

	A	COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **n**, an error condition is generated.
Output	**A**	The upper or lower triangle of the updated **A** matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **A** is unchanged.

$$A \leftarrow \alpha x x^* + \beta A^* \quad \text{with} \quad A=A^*$$

Name F_CHER2/F_ZHER2
Hermitian rank-2 update

Purpose F_xHER2 performs the Hermitian rank-2 update

$$A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + \beta A \quad \text{with} \quad A=A^*$$

where A is a complex Hermitian matrix, α is a complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are complex n-vectors, and x^* and y^* are the conjugate transposes of x and y, respectively. β is a real scalar. The routine returns immediately if n is less than or equal to zero.

Refer to "SSYR2/DSYR2/CHER2/ZHER2" on page 211 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
INTEGER          INCX, INCY, LDA, N, UPLO
REAL*4           BETA
COMPLEX*8        ALPHA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER          INCX, INCY, LDA, N, UPLO
REAL*8           BETA
COMPLEX*16       ALPHA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)
```

Input

UPLO	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
N	Number of elements of vector x.
ALPHA	The scalar ALPHA.
X	COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	The scalar BETA.
	A	COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **n**, an error condition is generated.
Output	**A**	The upper or lower triangle of the updated *A* matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **A** is unchanged.

$$A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + \beta A \quad \text{with} \quad A=A^*$$

Name

F_CHPMV/F_ZHPMV
Hermitian packed matrix-vector multiply

Purpose

F_xHPMV multiplies a vector x by a Hermitian packed matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero, or if β is equal to one and α is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

Refer to "SSPMV/DSPMV/CHPMV/ZHPMV" on page 187 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array. Refer to "SSPMV/DSPMV/CHPMV/ZHPMV" on page 187 for an example of the packed storage of symmetric or Hermitian matrices.

Usage

```
INTEGER      INCX, INCY, N, UPLO
COMPLEX*8    ALPHA, BETA
COMPLEX*8    AP( * ), X( * ), Y( * )
SUBROUTINE F_CHPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y,
INCY)

INTEGER      INCX, INCY, N, UPLO
COMPLEX*16   ALPHA, BETA
COMPLEX*16   AP( * ), X( * ), Y( * )
SUBROUTINE F_ZHPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y,
INCY)
```

Input

UPLO	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
ALPHA	The scalar ALPHA.
AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n complex Hermitian matrix A, stored by columns in packed form.

	X	COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	Y	COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with A=A*

Name F_CHPR/F_ZHPR
Hermitian rank-1 update

Purpose F_xHPR performs the Hermitian rank-1 update

$$A \leftarrow \alpha x x^* + \beta A^* \quad \text{with} \quad A=A^*$$

where A is an Hermitian matrix stored in packed form, α and β are real scalars, x is a complex n-vector, and x^* is the conjugate transpose of x.

Refer to "SSPR/DSPR/CHPR/ZHPR" on page 191 for a description of the equivalent HP MLIB legacy BLAS subprograms and an illustration of the packed storage of symmetric or Hermitian matrices.

Matrix Storage Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array (refer to the *AP* matrix).

Usage

```
INTEGER          INCX, N, UPLO
REAL*4           ALPHA, BETA
COMPLEX*8        AP( * ), X( * )
SUBROUTINE F_CHPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER          INCX, N, UPLO
REAL*8           ALPHA, BETA
COMPLEX*16       AP( * ), X( * )
SUBROUTINE F_ZHPR (UPLO, N, ALPHA, X, INCX, BETA, AP)
```

Input

UPLO Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.

N Number of elements of vector x.

ALPHA REAL scalar ALPHA.

X COMPLEX array, minimum length
(**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

	BETA	REAL scalar BETA.
	AP	Complex array, dimension (LDA, N). Contains the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
Output	**AP**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input. $A \leftarrow \alpha x x^* + \beta A^*$ with A=A*

Name F_CHPR2/F_ZHPR2
Hermitian rank-2 update

Purpose F_xHPR2 performs the Hermitian rank-2 update

$$A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + \beta A \quad \text{with} \quad A=A^*$$

where A is an n-by-n complex Hermitian matrix stored in packed form, α is a complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are complex n-vectors, and x^* and y^* are the conjugate transposes of x and y, respectively. This routine returns immediately if n is less than or equal to zero.

Refer to "SSPR2/DSPR2/CHPR2/ZHPR2" on page 195 for a description of the equivalent HP MLIB legacy BLAS subprograms and an illustration of the packed storage of symmetric or Hermitian matrices.

Matrix Storage Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array (refer to the *AP* matrix).

Usage

```
INTEGER          INCX, INCY, LDA, N, UPLO
REAL*4           BETA
COMPLEX*8        ALPHA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)

INTEGER          INCX, INCY, LDA, N, UPLO
REAL*8           BETA
COMPLEX*16       ALPHA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)
```

Input

UPLO Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.

N Number of elements of vector x.

ALPHA REAL scalar ALPHA.

X COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component $x(i)$, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

	Y	COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component *y(i)*, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then (*i*) is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then (*i*) is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	COMPLEX scalar BETA.
	AP	Complex array, dimension (LDA, N). Contains the upper or lower triangle, as specified by **uplo** of an *n*-by-*n* real symmetric or complex Hermitian matrix A, stored by columns in packed form.
Output	**AP**	The upper or lower triangle of the updated *A* matrix, as specified by **uplo**, replaces the input. $A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + \beta A$ with A=A*

Name F_SFPINFO/F_DFPINFO
Environmental inquiry

Purpose F_xFPINFO queries for machine-specific floating point characteristics. For BLAS Standard routines, error bounds and limitations due to overflow and underflow depend on details of how floating point numbers are represented. These details are available by calling F_xFPINFO.

Usage

```
INTEGER      CMACH
REAL*4       FUNCTION F_SFPINFO (CMACH)

INTEGER      CMACH
REAL*4       FUNCTION F_DFPINFO (CMACH)
```

Input **CMACH**

A named integer constant. The names for the **CMACH** argument are given in the appropriate language's include file.

Refer to Table 2-1 on page 131 for a description of **CMACH** floating point parameters, the corresponding BLAS Standard named constant (from the Fortran 77 include file), and the values returned by F_xFPINFO.

Name

F_SGBMV/F_DGBMV/F_CGBMV/F_ZGBMV
General band matrix-vector multiply

Purpose

F_xGBMV multiplies a vector x by a general band matrix A, its transpose, or its conjugate transpose, scales the resulting vector, and adds it to the scaled vector operand y. If m or n is less than or equal to zero, or if β is equal to one and α is equal to zero, the routine returns immediately. If *kl* or *ku* is less than zero, or if *lda* is less than *kl* + *ku* + *1*, an error flag is set and passed to the error handler. The matrix-vector multiply can be defined as one of the following:

$$y \leftarrow \alpha Ax + \beta y$$

$$y \leftarrow \alpha A^T x + \beta y$$

$$y \leftarrow \alpha A^* x + \beta y$$

Refer to "SGBMV/DGBMV/CGBMV/ZGBMV" on page 159 for a description of the equivalent HP MLIB legacy BLAS subprograms and an example of the storage of general band matrices.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, you need only provide the elements within the band of A. The subprograms for general band matrices use less storage than the subprograms for general full matrices if $kl+ku < n$.

Usage

```
INTEGER      INCX, INCY, KL, KU, LDA, M, N, TRANS
REAL*4       ALPHA, BETA
REAL*4       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
INCX, BETA, Y, INCY)

INTEGER      INCX, INCY, KL, KU, LDA, M, N, TRANS
REAL*8       ALPHA, BETA
REAL*8       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
INCX, BETA, Y, INCY)

INTEGER      INCX, INCY, KL, KU, LDA, M, N, TRANS
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
INCX, BETA, Y, INCY)
```

```
INTEGER      INCX, INCY, KL, KU, LDA, M, N, TRANS
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
INCX, BETA, Y, INCY)
```

Input

TRANS Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following:

BLAS_NO_TRANS, BLAS_TRANS, BLAS_CONJ_TRANS

M Number of rows in matrix A, $m > 0$. If $m \leq 0$, the subprograms do not reference A, X, or Y.

N Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.

KL The lower bandwidth of A, that is, the number of nonzero diagonals below the principal diagonal in the band, $0 \leq KL < n$.

KU The upper bandwidth of A, that is, the number of nonzero diagonals above the principal diagonal in the band, $0 \leq KU < n$.

ALPHA The scalar ALPHA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.

A REAL or COMPLEX array, dimension (LDA, N).

LDA Leading dimension of array A. If **lda** < (**kl** + **ku** + 1), an error condition is generated.

X REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

INCX Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.

BETA The scalar BETA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.

Y REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.

	INCY	Increment for the array y. A vector y having component y_i, $i = 1,..., n$, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $$y \leftarrow \alpha Ax + \beta y$$ where A can be A, A^T, or A*.

Name

F_SGE_COPY/F_DGE_COPY/F_CGE_COPY/F_ZGE_COPY
Matrix copy

Purpose

F_xGE_COPY copies an *m-by-n* matrix A, its transpose A^T, or its conjugate transpose A*, and stores the result in a matrix B.

$$B \leftarrow A, \quad B \leftarrow A^T \text{ or } \quad B \leftarrow A^*$$

Matrices *A* and *B* have the same storage format.

Refer to "SGECPY/DGECPY/CGECPY/ZGECPY" on page 165 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage

```
INTEGER          LDA, LDB, M, N, TRANS
REAL*4           A( LDA, * ), B( LDB, * )
SUBROUTINE F_SGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER          LDA, LDB, M, N, TRANS
REAL*8           A( LDA, * ), B( LDB, * )
SUBROUTINE F_DGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER          LDA, LDB, M, N, TRANS
COMPLEX*8        A( LDA, * ), B( LDB, * )
SUBROUTINE F_CGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER          LDA, LDB, M, N, TRANS
COMPLEX*16       A( LDA, * ), B( LDB, * )
SUBROUTINE F_ZGE_COPY(TRANS, M, N, A, LDA, B, LDB)
```

Input

TRANS Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following constants:

BLAS_NO_TRANS

BLAS_TRANS

BLAS_CONJ_TRANS

M Number of rows in matrix B, **m** ≥ 0. If **m** = 0, the subprograms do not reference **A** or **B**.

N Number of columns in matrix *B*, **n** ≥ 0. If **n** = 0, the subprograms do not reference **A** or **B**.

A Array containing the *m-by-n* matrix A.

	LDA	Leading dimension of array A. If **lda** < 1 or **lda** < **m** an error flag is set and passed to the error handler
	B	Array containing the m by n matrix B.
	LDB	Leading dimension of array B. Under the following conditions an error flag is set and passed to the error handler: • **TRANS** = **BLAS_NO_TRANS**, and **lda** < 1 or **lda** < **m** • **TRANS** = **BLAS_TRANS**, and **lda** < 1 or **lda** < **n** • **TRANS** = **BLAS_CONJ_TRANS**, and **lda** < 1 or **lda** < **n**
Output	**B**	Array containing the m-by-n matrix B copied *from* A, A^T, or A*.

Name F_SGE_TRANS/F_DGE_TRANS/F_CGE_TRANS/F_ZGE_TRANS
Matrix transposition

Purpose F_xGE_TRANS performs the matrix transposition, or conjugate transposition, of a square matrix A. F_xGE_TRANS returns immediately if n is less than or equal to zero.

$$A \leftarrow A^T \quad \text{or} \quad A \leftarrow A^*$$

Usage

```
INTEGER          CONJ, LDA,N
REAL*4           A( LDA, * )
SUBROUTINE F_SGE_TRANS(CONJ, N, A, LDA)

INTEGER          CONJ, LDA,N
REAL*8           A( LDA, * )
SUBROUTINE F_DGE_TRANS(CONJ, N, A, LDA)

INTEGER          CONJ, LDA,N
COMPLEX*8        A( LDA, * )
SUBROUTINE F_CGE_TRANS(CONJ, N, A, LDA)

INTEGER          CONJ, LDA,N
COMPLEX*16       A( LDA, * )
SUBROUTINE F_ZGE_TRANS(CONJ, N, A, LDA)
```

Input

CONJ	Specifies conjugation. Use either **BLAS_CONJ** or **BLAS_NO_CONJ**. When A is real the **conj** operator argument has no effect.
N	Number of rows and columns in matrix A, $n > 0$.
A	Array containing the m-by-n matrix A.
LDA	Leading dimension of array A. When **lda** < 1 or **lda** < **n**, an error flag is set and passed to the error handler.

Output

A	The transposed m-by-n matrix replaces the input matrix A.

Name F_SGEMM/F_DGEMM/F_CGEMM/F_ZGEMM
General matrix-matrix multiply

Purpose F_xGEMM performs the general matrix-matrix multiply

$$C \leftarrow \alpha op(A) op(B) + \beta C$$

where α and β are scalars, and A, B, and C are general matrices. op(A) and op(B) denote A, A^T, or A* and B, B^T, or B*, respectively.

This F_xGEMM interface encompasses the legacy BLAS routine xGEMM with added functionality for band matrix-matrix multiplication. Refer to "SGEMM/DGEMM/CGEMM/ZGEMM" on page 167 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage

```
INTEGER      K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
REAL*4       ALPHA, BETA
REAL*4       A( LDA, * ), B( LDB, * ), C( LDC, * )
SUBROUTINE F_SGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
B, LDB, BETA, C, LDC)

INTEGER      K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
REAL*8       ALPHA, BETA
REAL*8       A( LDA, * ), B( LDB, * ), C( LDC, * )
SUBROUTINE F_DGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
B, LDB, BETA, C, LDC)

INTEGER      K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), B( LDB, * ), C( LDC, * )
SUBROUTINE F_CGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
B, LDB, BETA, C, LDC)

INTEGER      K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), B( LDB, * ), C( LDC, * )
SUBROUTINE F_ZGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
B, LDB, BETA, C, LDC)
```

Input

TRANS-(A/B)
Specifies whether to apply the matrix (A or B), its transpose (A^T or B^T), or its conjugate transpose (A* or B*). Use one of the following constants:

BLAS_NO_TRANS, BLAS_TRANS, BLAS_CONJ_TRANS

M
Number of rows in matrix *C*, **m** ≥ 0. If **m** = 0, the subprograms do not reference *A*, *B*, or *C*.

N
Number of columns in matrix *C*, **n** ≥ 0. If **n** = 0, the subprograms do not reference *A*, *B*, or *C*.

K
The *middle* dimension of the matrix multiply, **k** ≥ 0. If **k** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **A** or **B**.

ALPHA
The scalar ALPHA. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing *A* or *B*.

A
Array containing the matrix *A*, whose size is indicated by **TRANSA**:

BLAS_NO_TRANS *m*-by-*k* matrix *A*

BLAS_TRANS *k*-by-*m* matrix A^T

BLAS_CONJ_TRANS *k*-by-*m* matrix *A**

LDA
Leading dimension of array *A*.

Error conditions for **lda** depend on the value of **transa**.

Each of the following conditions generates an error flag that is passed to the error handler:

- **lda** < 1
- **TRANSA** = **BLAS_NO_TRANS** and **lda** < *m*
- **TRANSA** = **BLAS_TRANS** and **lda** < *k*
- **TRANSA** = **BLAS_CONJ_TRANS** and **lda** < *k*

B
Array containing the matrix *B*, whose size is indicated by **TRANSB**:

BLAS_NO_TRANS *k*-by-*n* matrix *B*

BLAS_TRANS *n*-by-*k* matrix B^T

BLAS_CONJ_TRANS *n*-by-*k* matrix *B**

	LDB	Leading dimension of array *B*. Error conditions for **ldb** depend on the value of **transb**. Each of the following conditions generates an error flag that is passed to the error handler: • **ldb** < 1 • **TRANSB** = **BLAS_NO_TRANS** and **ldb** < k • **TRANSB** = **BLAS_TRANS** and **ldb** < n • **TRANSB** = **BLAS_CONJ_TRANS** and **ldb** < n
	BETA	The scalar BETA.
	C	The m-by-n matrix operand C. The representation of the matrix entry $c_{i,j}$ in C is denoted by $C(i, j)$ for all (i, j) in the interval [0... m - 1] x [0... n - 1].
	LDC	Leading dimension of array *C*. If **ldc** < 1 or **ldc** < m, an error flag is generated and passed to the error handler.
Output	**C**	The updated m-by-n matrix C replaces the input.

$$C \leftarrow \alpha op(A) op(B) + \beta C$$

Name F_SGEMV/F_DGEMV/F_CGEMV/F_ZGEMV
General matrix-vector multiply

Purpose F_xGEMV multiplies a vector x by a general matrix A, its transpose, or its conjugate transpose, scales the resulting vector, and adds it to the scaled vector operand y. If m or n is less than or equal to zero, or if β is equal to one and α is equal to zero, the routine returns immediately. If *lda* is less than one, or less than m, an error flag is set and passed to the error handler.The matrix-vector multiply can be defined as one of the following:

$$y \leftarrow \alpha Ax + \beta y$$

$$y \leftarrow \alpha A^T x + \beta y$$

$$y \leftarrow \alpha A^* x + \beta y$$

Refer to "SGEMV/DGEMV/CGEMV/ZGEMV" on page 175 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage

```
INTEGER      INCX, INCY, LDA, M, N, TRANS
REAL*4       ALPHA, BETA
REAL*4       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)

INTEGER      INCX, INCY, LDA, M, N, TRANS
REAL*8       ALPHA, BETA
REAL*8       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX,
BETA, Y, INCY)

INTEGER      INCX, INCY, LDA, M, N, TRANS
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX,
BETA, Y, INCY)

INTEGER      INCX, INCY, LDA, M, N, TRANS
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)
```

Input	**TRANS**	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS, BLAS_TRANS, BLAS_CONJ_TRANS.**
	M	Number of rows in matrix A, $m > 0$. If $m \leq 0$, the subprograms do not reference A, X, or Y.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	ALPHA	The scalar ALPHA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **m**, an error condition is generated.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated vector replaces the input. $$y \leftarrow \alpha Ax + \beta y$$ where A can be A, A^T, or A*.

Name F_SGEMVER/F_DGEMVER/F_CGEMVER/F_ZGEMVER
Multiple matrix-vector multiply, rank 2 update

Purpose F_xGEMVER precedes a combined matrix-vector and a transpose matrix-vector multiply with a rank two update.

$$\hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T$$

$$x \leftarrow \beta \hat{A}^T y + z$$

$$w \leftarrow \alpha \hat{A} x$$

Matrix A is updated by $u_1 v_1^T$ and $u_2 v_2^T$. The transpose of the updated matrix is multiplied by a vector y. The resulting vector is scaled and added to the vector operand z, and stored in x. The operand x is multiplied by the updated matrix A. The resulting vector is stored in w. If m or n is less than or equal to zero, this function returns immediately.

Usage

```
INTEGER      INCW, INCX, INCY, INCZ, LDA, M, N
REAL*4       A( LDA, * ), W( * ), X( * ), Y( * ), Z( * )
REAL*4       ALPHA, BETA
REAL*4       U1( * ), V1( * ), U2( * ), V2( * )
SUBROUTINE F_SGEMVER (M, N, A, LDA, U1, V1, U2, V2, ALPHA, X,
INCX, Y, INCY, BETA, W, INCW, Z, INCZ)

INTEGER      INCW, INCX, INCY, INCZ, LDA, M, N
REAL*8       A( LDA, * ), W( * ), X( * ), Y( * ), Z( * )
REAL*8       ALPHA, BETA
REAL*8       U1( * ), V1( * ), U2( * ), V2( * )
SUBROUTINE F_DGEMVER (M, N, A, LDA, U1, V1, U2, V2, ALPHA, X,
INCX, Y, INCY, BETA, W, INCW, Z, INCZ)

INTEGER      INCW, INCX, INCY, INCZ, LDA, M, N
COMPLEX*8    A( LDA, * ), W( * ), X( * ), Y( * ), Z( * )
COMPLEX*8    ALPHA, BETA
COMPLEX*8    U1( * ), V1( * ), U2( * ), V2( * )
SUBROUTINE F_CGEMVER (M, N, A, LDA, U1, V1, U2, V2, ALPHA, X,
INCX, Y, INCY, BETA, W, INCW, Z, INCZ)

INTEGER      INCW, INCX, INCY, INCZ, LDA, M, N
COMPLEX*16   A( LDA, * ), W( * ), X( * ), Y( * ), Z( * )
COMPLEX*16   ALPHA, BETA
COMPLEX*16   U1( * ), V1( * ), U2( * ), V2( * )
SUBROUTINE F_ZGEMVER (M, N, A, LDA, U1, V1, U2, V2, ALPHA, X,
INCX, Y, INCY, BETA, W, INCW, Z, INCZ)
```

Input

M	Number of rows in matrix A, $m > 0$. If $m \leq 0$, the subprograms do not reference A, X, or Y.
N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
A	REAL or COMPLEX array, dimension (LDA, N).
LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **m**, an error flag is set and passed to the error handler.
U1	REAL or COMPLEX array, dimension M.
V1	REAL or COMPLEX array, dimension N.
U2	REAL or COMPLEX array, dimension M.
V2	REAL or COMPLEX array, dimension N.
ALPHA	The scalar ALPHA.
INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
BETA	The scalar BETA.
INCW	Increment for the array w. A vector w having component w_i, i = 1,..., n, is stored in an array **W**() with increment argument **incw**. If **incw** > 0 then w_i is stored in **W** (1 + (i - 1) x **incw**). If **incw** < 0 then w_i is stored in **W** (1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
Z	REAL or COMPLEX array, minimum length (**N** - 1) x |**incz**| + 1.
INCZ	Increment for the array y. A vector z having component z_i, i = 1,..., n, is stored in an array **Z**() with increment argument **incw**. If **incw** > 0 then z_i is stored in **Z**(1 + (i - 1) x **incw**). If **incw** < 0 then z_i is stored in **Z**(1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.

Output

X Contains the result after A is first updated by $u_1 v_1^T$ and $u_2 v_2^T$, then the transpose of the updated matrix is multiplied by vector y.

$$\hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T$$

$$x \leftarrow \beta \hat{A}^T y + z$$

REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.

W Stores the resulting vector when the operand x is multiplied by the updated matrix A.

$$w \leftarrow \alpha \hat{A} x$$

REAL or COMPLEX array, minimum length (**N** - 1) x |**incw**| + 1.

Name F_SGEMVT/F_DGEMVT/F_CGEMVT/F_ZGEMVT
Multiple matrix-vector multiply

Purpose F_xGEMVT combines a matrix-vector and a transposed matrix-vector multiply.

$$x \leftarrow \beta A^T y + z$$

$$w \leftarrow \alpha A x$$

Specifically, F_xGEMVT routines perform the following operations:

1. Multiply a vector y by a general matrix A^T.
2. Scale the resulting vector by β and store the result in the vector operand x.
3. Multiply the matrix by the resultant vector x.
4. Scale the resulting vector by α and store it in the vector operand w.

If m or n is less than or equal to zero, this function returns immediately.

Usage

```
INTEGER          INCX, INCY, LDA, M, N
REAL*4           ALPHA, BETA, A( LDA, * ), X( * ), Y( * ), W( * ), Z( * )
SUBROUTINE F_SGEMVT (M, N, ALPHA, A, LDA, X, INCX, Y, INCY,
BETA, W, INCW, Z, INCZ)

INTEGER          INCX, INCY, LDA, M, N
REAL*8           ALPHA, BETA, A( LDA, * ), X( * ), Y( * ), W( * ), Z( * )
SUBROUTINE F_DGEMVT (M, N, ALPHA, A, LDA, X, INCX, Y, INCY,
BETA, W, INCW, Z, INCZ)

INTEGER          INCX, INCY, LDA, M, N
COMPLEX*8        ALPHA, BETA, A( LDA, * ), X( * ), Y( * ), W( * ), Z( * )
SUBROUTINE F_CGEMVT (M, N, ALPHA, A, LDA, X, INCX, Y, INCY,
BETA, W, INCW, Z, INCZ)

INTEGER          INCX, INCY, LDA, M, N
COMPLEX*16       ALPHA, BETA, A( LDA, * ), X( * ), Y( * ), W( * ), Z( * )
SUBROUTINE F_ZGEMVT (M, N, ALPHA, A, LDA, X, INCX, Y, INCY,
BETA, W, INCW, Z, INCZ)
```

Input

M	Number of rows in matrix A, $m > 0$. If $m \leq 0$, the subprograms do not reference A, X, or Y.
N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
ALPHA	REAL or COMPLEX scalar ALPHA.
A	REAL or COMPLEX array, dimension (LDA, N).

	LDA	Leading dimension of array A. If **lda** < 1 or **lda** < **m**, an error flag is set and passed to the error handler.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	REAL or COMPLEX scalar BETA.
	INCW	Increment for the array w. A vector w having component w_i, i = 1,..., n, is stored in an array **W**() with increment argument **incw**. If **incw** > 0 then w_i is stored in **W** (1 + (i - 1) x **incw**). If **incw** < 0 then w_i is stored in **W** (1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
	Z	REAL or COMPLEX array, minimum length (**N** - 1) x |**incz**| + 1.
	INCZ	Increment for the array y. A vector z having component z_i, i = 1,..., n, is stored in an array **Z**() with increment argument **incw**. If **incw** > 0 then z_i is stored in **Z**(1 + (i - 1) x **incw**). If **incw** < 0 then z_i is stored in **Z**(1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
Output	**X**	Result of first matrix-vector multiplication and scaling by β. Refer to "Purpose" on page 288 for details. $$x \leftarrow \beta A^T y + z$$
	W	Result of second matrix-vector multiply. Refer to "Purpose" on page 288 for details. $$w \leftarrow \alpha A x$$

Name

F_SGER/F_DGER/F_CGER/F_ZGER
General rank-1 update

Purpose

F_xGER performs the following rank-1 operations:

$$\text{When } A \in IR^{n^2},\ A \leftarrow \alpha x y^T + \beta A$$

$$\text{When } A \in C^{n^2},\ A \leftarrow \alpha x y^T + \beta A \text{ or}$$

$$A \leftarrow \alpha x y^* + \beta A$$

where A is an m-by-n matrix, α and β are scalars, x is an m-vector, y is an n-vector, and y^T and y^* are the transpose and conjugate transpose of y, respectively. The operator argument **CONJ** is only referenced when x and y are complex vectors.

When x and y are complex vectors, the vector components y_i are used unconjugated or conjugated as specified by the operator argument **CONJ**.

Refer to "SGER/DGER/CGERC/CGERU/ZGERC/ZGERU" on page 179 for a description of HP MLIB legacy BLAS subprograms for rank-1 update.

Usage

```
INTEGER         CONJ, INCX, INCY, LDA, M, N
REAL*4          ALPHA, BETA
REAL*4          A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SGER (CONJ, M, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER         CONJ, INCX, INCY, LDA, M, N
REAL*8          ALPHA, BETA
REAL*8          A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DGER (CONJ, M, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER         CONJ, INCX, INCY, LDA, M, N
COMPLEX*8       ALPHA, BETA
COMPLEX*8       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CGER (CONJ, M, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER         CONJ, INCX, INCY, LDA, M, N
COMPLEX*16      ALPHA, BETA
COMPLEX*16      A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZGER (CONJ, M, N, ALPHA, X, INCX, Y, INCY, BETA,A,
LDA)
```

Input	**CONJ**	Specifies conjugation for vector components in complex routines. Vector components are used conjugated or unconjugated. Use either **BLAS_CONJ** or **BLAS_NO_CONJ**. When x and y are real vectors the **conj** operator argument has no effect.
	M	Number of rows in matrix A, $m > 0$. If $m \leq 0$, the subprograms do not reference A, X, or Y.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	The scalar BETA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array A. **lda** < 1 and **lda** < **m** are illegal values.
Output	**A**	The updated A matrix replaces the input.

Name

F_SSBMV/F_DSBMV/F_CSBMV/F_ZSBMV
Symmetric band matrix-vector multiply

Purpose

F_xSBMV multiplies a vector x by a real or complex symmetric band matrix A, scales the resulting vector, and adds it to the scaled vector operand y. If n is less than or equal to zero, or if β is equal to one and α is equal to zero, this routine returns immediately.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^T$$

Refer to "SSBMV/DSBMV/CHBMV/ZHBMV" on page 182 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and because either triangle of A can be obtained from the other, you only need to provide the band within one triangle of A. Compared to storing the entire matrix, this can save memory in two ways: only the elements within the band are stored and of them only the upper or the lower triangle. Refer to "SSBMV/DSBMV/CHBMV/ZHBMV" on page 182 for an example of the storage of symmetric band matrices.

Usage

```
INTEGER      INCX, INCY, K, LDA, N, UPLO
REAL*4       ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)

INTEGER      INCX, INCY, K, LDA, N, UPLO
REAL*8       ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)

INTEGER      INCX, INCY, K, LDA, N, UPLO
COMPLEX*8    ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)

INTEGER      INCX, INCY, K, LDA, N, UPLO
COMPLEX*16   ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
Y, INCY)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	K	The number of non zero diagonals above or below the principal diagonal.
	ALPHA	The scalar ALPHA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **k**+1, an error condition is generated.
	X	REAL or COMPLEX array, minimum length **N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with $A=A^T$

Name

F_SSPMV/F_DSPMV/F_CSPMV/F_ZSPMV
Symmetric packed matrix-vector multiply

Purpose

F_xSPMV multiplies a vector x by a real or complex symmetric packed matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero, or if β is equal to one and α is equal to zero, this routine returns immediately.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^T$$

Refer to "SSPMV/DSPMV/CHPMV/ZHPMV" on page 187 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array. Refer to "SSPMV/DSPMV/CHPMV/ZHPMV" on page 187 for an example of packed storage for symmetric or Hermitian matrices.

Usage

```
INTEGER      INCX, INCY, N, UPLO
REAL*4       ALPHA, BETA
REAL*4       AP( * ), X( * ), Y( * )
SUBROUTINE F_SSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)

INTEGER      INCX, INCY, N, UPLO
REAL*8       ALPHA, BETA
REAL*8       AP( * ), X( * ), Y( * )
SUBROUTINE F_DSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y,
INCY)

INTEGER      INCX, INCY, N, UPLO
COMPLEX*8    ALPHA, BETA
COMPLEX*8    AP( * ), X( * ), Y( * )
SUBROUTINE F_CSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y,
INCY)

INTEGER      INCX, INCY, N, UPLO
COMPLEX*16   ALPHA, BETA
COMPLEX*16   AP( * ), X( * ), Y( * )
SUBROUTINE F_ZSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	ALPHA	REAL or COMPLEX scalar ALPHA.
	AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	REAL or COMPLEX scalar BETA.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with $A=A^T$

Name F_SSPR/F_DSPR/F_CSPR/F_ZSPR
Symmetric packed rank-1 update

Purpose F_xSPR performs the symmetric rank-1 update

$$A \leftarrow \alpha x x^T + \beta A \quad \text{with} \quad A=A^T$$

where A is an n-by-n real symmetric matrix stored in packed form, α and β are real or complex scalars, x is a real or complex n-vector, and x^T is the transpose of x. The routine returns immediately if n is less than or equal to zero.

This F_xSPR interface encompasses the legacy BLAS routine SSPR with added functionality for complex symmetric matrices. Refer to "SSPR/DSPR/CHPR/ZHPR" on page 191 for a description of the equivalent HP MLIB legacy BLAS subprograms and an illustration of the packed storage of symmetric or Hermitian matrices.

Matrix Storage Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a one-dimensional array (refer to the *AP* matrix).

Usage

```
INTEGER          INCX, N, UPLO
REAL*4           ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_SSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER          INCX, N, UPLO
REAL*8           ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_DSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER          INCX, N, UPLO
COMPLEX*8        ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_CSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER          INCX, N, UPLO
COMPLEX*16       ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_ZSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
Output	**AP**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input. $A \leftarrow \alpha x x^T + \beta A$ with $A=A^T$

Name

F_SSPR2/F_DSPR2/F_CSPR2/F_ZSPR2
Symmetric rank-2 update

Purpose

F_xSPR2 performs the symmetric rank-2 update

$$A \leftarrow \alpha x y^T + \bar{\alpha} y x^T + \beta A \quad \text{with} \quad A=A^T$$

where A is an n-by-n real symmetric matrix stored in packed form, α and β are real or complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are real or complex n-vectors, and x^T and y^T are transposes of x and y, respectively.

This F_xSPR2 interface encompasses the legacy BLAS routine SSPR2 with added functionality for complex symmetric matrices. Refer to "SSPR2/DSPR2/CHPR2/ZHPR2" on page 195 for a description of the HP MLIB legacy BLAS subprograms and an illustration of the packed storage of symmetric or Hermitian matrices.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A, either the upper or the lower triangle. Compared to storing the entire matrix, you save memory by supplying that triangle stored column-by-column in packed form in a 1-dimensional array (refer to the *AP* matrix).

Usage

```
INTEGER         INCX, INCY, N, UPLO
REAL*4          ALPHA, BETA
REAL*4          AP( * ), X( * ), Y( * )
SUBROUTINE F_SSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)

INTEGER         INCX, INCY, N, UPLO
REAL*8          ALPHA, BETA
REAL*8          AP( * ), X( * ), Y( * )
SUBROUTINE F_DSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)

INTEGER         INCX, INCY, N, UPLO
COMPLEX*8       ALPHA, BETA
COMPLEX*8       AP( * ), X( * ), Y( * )
SUBROUTINE F_CSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)

INTEGER         INCX, INCY, N, UPLO
COMPLEX*16      ALPHA, BETA
COMPLEX*16      AP( * ), X( * ), Y( * )
SUBROUTINE F_ZSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, AP)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (N - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (N - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	The scalar BETA.
	AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
Output	**AP**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input. $$A \leftarrow \alpha x y^T + \bar{\alpha} y x^T + \beta A \quad \text{with} \quad A=A^T$$

Name

F_SSYMV/F_DSYMV/F_CSYMV/F_ZSYMV
Symmetric matrix-vector multiply

Purpose

F_xSYMV multiplies a vector x by a real or complex symmetric matrix A, scales the resulting vector, and adds it to the scaled vector operand y. If n is less than or equal to zero, or if β is equal to one and α is equal to zero, this routine returns immediately.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^T$$

Refer to "SSYMV/DSYMV/CHEMV/ZHEMV" on page 204 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, you only need to provide one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The other triangle of the array is not referenced.

Usage

```
INTEGER          INCX, INCY, LDA, N, UPLO
REAL*4           ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)

INTEGER          INCX, INCY, LDA, N, UPLO
REAL*8           ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)

INTEGER          INCX, INCY, LDA, N, UPLO
COMPLEX*8        ALPHA, BETA, A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)

INTEGER          INCX, INCY, LDA, N, UPLO
COMPLEX*16       ALPHA, BETA,A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
INCY)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	ALPHA	The scalar ALPHA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **n** an error condition is generated.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA. If **beta** = 1 and **alpha** = 0, this routine returns immediately.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array *y*. A vector *y* having component y_i, *i* = 1,..., *n*, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (*i* - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
Output	**Y**	The updated **Y** vector replaces the input. $y \leftarrow \alpha Ax + \beta y$ with $A=A^T$

Name

F_SSYR/F_DSYR/F_CSYR/F_ZSYR
Symmetric rank-1 update

Purpose

F_xSYR performs the symmetric rank-1 update

$$A \leftarrow \alpha x x^T + \beta A \quad \text{with} \quad A=A^T$$

where A is an n-by-n real symmetric matrix, α and β are real or complex scalars, x is a real or complex n-vector, and x^T is the transpose of x. The routine returns immediately if n is less than or equal to zero.

This F_xSYR interface encompasses the legacy BLAS routine SSYR with added functionality for complex symmetric matrices. Refer to "SSYR/DSYR/CHER/ZHER" on page 208 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
INTEGER       INCX, LDA, N, UPLO
REAL*4        ALPHA, BETA
REAL*4        A( LDA, * ), X( * )
SUBROUTINE F_SSYR (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)

INTEGER       INCX, LDA, N, UPLO
REAL*8        ALPHA, BETA
REAL*8        A( LDA, * ), X( * )
SUBROUTINE F_DSYR (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)

INTEGER       INCX, LDA, N, UPLO
COMPLEX*8     ALPHA, BETA
COMPLEX*8     A( LDA, * ), X( * )
SUBROUTINE F_CSYR (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)

INTEGER       INCX, LDA, N, UPLO
COMPLEX*16    ALPHA, BETA
COMPLEX*16    A( LDA, * ), X( * )
SUBROUTINE F_ZSYR (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, dimension (1 + (**N** - 1) x **incx**).
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	BETA	The scalar BETA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array A. **lda** < 1 and **lda** < **n** are illegal values.
Output	**A**	The upper or lower triangle of the updated **A** matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **A** is unchanged. $A \leftarrow \alpha x x^T + \beta A$ with $A=A^T$

Name

F_SSYR2/F_DSYR2/F_CSYR2/F_ZSYR2
Symmetric rank-2 update

Purpose

F_xSYR2 performs the symmetric rank-2 update

$$A \leftarrow \alpha x y^T + \bar{\alpha} y x^T + \beta A \quad \text{with} \quad A=A^T$$

where A is an n-by-n real symmetric matrix, α and β are real or complex scalars, $\bar{\alpha}$ is the complex conjugate of α, x and y are real or complex n-vectors, and x^T and y^T are the transposes of x and y, respectively.

This F_xSYR2 interface encompasses the legacy BLAS routine SSYR2 with added functionality for complex symmetric matrices. Refer to "SSYR2/DSYR2/CHER2/ZHER2" on page 211 for a description of the HP MLIB legacy BLAS subprograms.

Matrix Storage

Because either triangle of A can be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix, and the same triangle of the updated matrix is returned in the array. The other triangle of the array is not referenced.

Usage

```
INTEGER      INCX, INCY, LDA, N, UPLO
REAL*4       ALPHA, BETA
REAL*4       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_SSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER      INCX, INCY, LDA, N, UPLO
REAL*8       ALPHA, BETA
REAL*8       A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_DSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER      INCX, INCY, LDA, N, UPLO
COMPLEX*8    ALPHA, BETA
COMPLEX*8    A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_CSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)

INTEGER      INCX, INCY, LDA, N, UPLO
COMPLEX*16   ALPHA, BETA
COMPLEX*16   A( LDA, * ), X( * ), Y( * )
SUBROUTINE F_ZSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
LDA)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	N	Number of elements of vector x.
	ALPHA	The scalar ALPHA.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_i is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	BETA	The scalar BETA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array A. If **lda** < 1 or **lda** < **n**, an error condition is generated.
Output	**A**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **A** is unchanged. $A \leftarrow \alpha x y^T + \bar{\alpha} y x^T + \beta A$ with $A=A^T$

Name

F_STBMV/F_DTBMV/F_CTBMV/F_ZTBMV
Triangular banded matrix-vector multiply

Purpose

F_xTBMV multiplies a vector x by a banded triangular matrix *(T)*, its transpose (T^T), or its conjugate transpose (T*), and copies the resulting vector to the vector operand x. If n is less than or equal to zero, this routine returns immediately.

$$x \leftarrow \alpha T x$$

$$x \leftarrow \alpha T^T x$$

$$x \leftarrow \alpha T^* x$$

Refer to "STBMV/DTBMV/CTBMV/ZTBMV" on page 223 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that can be nonzero. Refer to the examples in "STBMV/DTBMV/CTBMV/ZTBMV" on page 223 for information about the storage of triangular band matrices.

Usage

```
INTEGER      DIAG, INCX, K, LDA, N, TRANS, UPLO
REAL*4       ALPHA
REAL*4       A( LDA, * ), X( * )
SUBROUTINE F_STBMV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA, X,
INCX)

INTEGER      DIAG, INCX, K, LDA, N, TRANS, UPLO
REAL*8       ALPHA
REAL*8       A( LDA, * ), X( * )
SUBROUTINE F_DTBMV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA,
X, INCX)

INTEGER      DIAG, INCX, K, LDA, N, TRANS, UPLO
COMPLEX*8    ALPHA
COMPLEX*8    A( LDA, * ), X( * )
SUBROUTINE F_CTBMV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA,
X, INCX)

INTEGER      DIAG, INCX, K, LDA, N, TRANS, UPLO
COMPLEX*16   ALPHA
COMPLEX*16   A( LDA, * ), X( * )
SUBROUTINE F_ZTBMV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA, X,
INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS** **BLAS_TRANS** **BLAS_CONJ_TRANS**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.
	N	Number of rows and columns in matrix A, and elements of vector X. $n > 0$. If $n \leq 0$, the subprograms do not reference A or X.
	K	The number of non zero diagonals above or below the principal diagonal.
	ALPHA	The scalar ALPHA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < **k** + 1, an error condition is generated.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (N - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**X**	The updated **X** vector replaces the input.

Name

F_STBSV/F_DTBSV/F_CTBSV/F_ZTBSV
Triangular banded solve

Purpose

F_xTBSV solves one of the following equations:

$$x \leftarrow \alpha T^{-1} x$$

$$x \leftarrow \alpha T^{-T} x$$

$$x \leftarrow \alpha T^{-*} x$$

where x is a vector and the matrix T is a unit, non-unit, upper, or lower triangular banded matrix, T^{-T} is the inverse of the transpose of T, and T^{-*} is the inverse of the conjugate transpose of T.

Refer to "STBSV/DTBSV/CTBSV/ZTBSV" on page 229 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **DIAG**), then the diagonal elements of the array also are not referenced.

Usage

```
INTEGER          DIAG, INCX, K, N, TRANS, UPLO
REAL*4           ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_STBSV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA, X,
INCX)

INTEGER          DIAG, INCX, K, N, TRANS, UPLO
REAL*8           ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_DTBSV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA, X,
INCX)

INTEGER          DIAG, INCX, K, N, TRANS, UPLO
COMPLEX*8        ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_CTBSV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA,  X,
INCX)

INTEGER          DIAG, INCX, K, N, TRANS, UPLO
COMPLEX*16       ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_ZTBSV (UPLO, TRANS, DIAG, N, K, ALPHA, A, LDA, X,
INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS, BLAS_TRANS, or BLAS_CONJ_TRANS.**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	K	The number of non zero diagonals above or below the principal diagonal.
	ALPHA	The scalar ALPHA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. **lda** < 1 and **lda** < **n** are illegal values.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**X**	The solution vector of the triangular system replaces the input.

Name

F_STPMV/F_DTPMV/F_CTPMV/F_ZTPMV
Triangular packed matrix-vector multiply

Purpose

F_xTPMV multiplies a vector x by a packed triangular matrix *(T)*, its transpose (T^T), or its conjugate transpose (T*), and copies the resulting vector to the vector operand x. If n is less than or equal to zero, this routine returns immediately.

$$x \leftarrow \alpha T x$$

$$x \leftarrow \alpha T^T x$$

$$x \leftarrow \alpha T^* x$$

Refer to "STPMV/DTPMV/CTPMV/ZTPMV" on page 235 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

You supply the upper or lower triangle of A, stored column-by-column in packed form in a 1-dimensional array. This saves memory compared to storing the entire matrix. Refer to "STPMV/DTPMV/CTPMV/ZTPMV" on page 235 for examples that illustrate the packed storage of a triangular matrix.

Usage

```
INTEGER      DIAG, INCX, N, TRANS, UPLO
REAL*4       ALPHA
REAL*4       AP( * ), X( * )
SUBROUTINE F_STPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER      DIAG, INCX, N, TRANS, UPLO
REAL*8       ALPHA
REAL*8       AP( * ), X( * )
SUBROUTINE F_DTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER      DIAG, INCX, N, TRANS, UPLO
COMPLEX*8    ALPHA
COMPLEX*8    AP( * ), X( * )
SUBROUTINE F_CTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER      DIAG, INCX, N, TRANS, UPLO
COMPLEX*16   ALPHA
COMPLEX*16   AP( * ), X( * )
SUBROUTINE F_ZTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS** **BLAS_TRANS** **BLAS_CONJ_TRANS**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.
	N	Number of rows and columns in matrix A, and elements of vector X. $n > 0$. If $n \leq 0$, the subprograms do not reference A or X.
	ALPHA	The scalar ALPHA.
	AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**X**	The updated **X** vector replaces the input.

Name

F_STPSV/F_DTPSV/F_CTPSV/F_ZTPSV
Triangular packed solve

Purpose

F_xTPSV solves one of the following equations:

$$x \leftarrow \alpha T^{-1} x$$

$$x \leftarrow \alpha T^{-T} x$$

$$x \leftarrow \alpha T^{-*} x$$

where x is a vector and the matrix T is a unit, non-unit, upper, or lower triangular packed matrix. T^{-T} is the inverse of the transpose of T, and T^{-*} is the inverse of the conjugate transpose of T.

Refer to "STPSV/DTPSV/CTPSV/ZTPSV" on page 239 for details of the HP MLIB legacy BLAS triangular-solve subprograms, and a description of packed storage for a triangular matrix.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **DIAG**), then the diagonal elements of the array also are not referenced.

Usage

```
INTEGER          DIAG, INCX, N, TRANS, UPLO
REAL*4           ALPHA, AP( * ), X( * )
SUBROUTINE F_STPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
REAL*8           ALPHA, AP( * ), X( * )
SUBROUTINE F_DTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
COMPLEX*8        ALPHA, AP( * ), X( * )
SUBROUTINE F_CTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
COMPLEX*16       ALPHA, AP( * ), X( * )
SUBROUTINE F_ZTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS, BLAS_TRANS, or BLAS_CONJ_TRANS.**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A, X, or Y.
	ALPHA	The scalar ALPHA.
	AP	Array containing the upper or lower triangle, as specified by **uplo** of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in packed form.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**AP**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input.

Name

F_STRMV/F_DTRMV/F_CTRMV/F_ZTRMV
Triangular matrix-vector multiply

Purpose

F_xTRMV multiplies a vector x by a general triangular matrix *(T)*, its transpose (T^T), or its conjugate transpose (T*), and copies the resulting vector to the vector operand x. If n is less than or equal to zero, this routine returns immediately.

$$x \leftarrow \alpha T x$$

$$x \leftarrow \alpha T^T x$$

$$x \leftarrow \alpha T^* x$$

Refer to "STRMV/DTRMV/CTRMV/ZTRMV" on page 247 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **DIAG**), then the diagonal elements of the array also is not referenced.

Usage

```
INTEGER       DIAG, INCX, LDA, N, TRANS, UPLO
REAL*4        ALPHA
REAL*4        A( LDA, * ), X( * )
SUBROUTINE F_STRMV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER       DIAG, INCX, LDA, N, TRANS, UPLO
REAL*8        ALPHA
REAL*8        A( LDA, * ), X( * )
SUBROUTINE F_DTRMV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER       DIAG, INCX, LDA, N, TRANS, UPLO
COMPLEX*8     ALPHA
COMPLEX*8     A( LDA, * ), X( * )
SUBROUTINE F_CTRMV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER       DIAG, INCX, LDA, N, TRANS, UPLO
COMPLEX*16    ALPHA
COMPLEX*16    A( LDA, * ), X( * )
SUBROUTINE F_ZTRMV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS** **BLAS_TRANS** **BLAS_CONJ_TRANS**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.
	N	Number of rows and columns in matrix A, and elements of vector X. $n > 0$. If $n \leq 0$, the subprograms do not reference A or X.
	ALPHA	The scalar ALPHA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. If **lda** < 1 or **lda** < **n**, an error condition is generated.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array *x*. A vector *x* having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (N - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**X**	The updated **X** vector replaces the input.

Name

F_STRMVT/F_DTRMVT/F_CTRMVT/F_ZTRMVT
Multiple triangular matrix-vector multiply

Purpose

F_xTRMVT combines a matrix-vector and a transpose matrix-vector multiply.

$$x \leftarrow T^T y$$

$$w \leftarrow Tz$$

F_xTRMVT multiplies a vector y by a triangular matrix T^T, storing the result as x. It also multiplies the matrix by the vector z, storing the result as w. If n is less than or equal to zero, this function returns immediately.

Usage

```
INTEGER          INCW, INCX, INCY, INCZ, LDT, N, UPLO
REAL*4           T( LDT, * ), W( * ), X( * ), Y( * ), Z( * )
SUBROUTINE F_STRMVT (UPLO, N, T, LDT, X, INCX, Y, INCY, W, INCW,
Z, INCZ)

INTEGER          INCW, INCX, INCY, INCZ, LDT, N, UPLO
REAL*8           T( LDT, * ), W( * ), X( * ), Y( * ), Z( * )
SUBROUTINE F_DTRMVT (UPLO, N, T, LDT, X, INCX, Y, INCY, W, INCW,
Z, INCZ)

INTEGER          INCW, INCX, INCY, INCZ, LDT, N, UPLO
COMPLEX*8        T( LDT, * ), W( * ), X( * ), Y( * ), Z( * )
SUBROUTINE F_CTRMVT (UPLO, N, T, LDT, X, INCX, Y, INCY, W, INCW,
Z, INCZ)

INTEGER          INCW, INCX, INCY, INCZ, LDT, N, UPLO
COMPLEX*16       T( LDT, * ), W( * ), X( * ), Y( * ), Z( * )
SUBROUTINE F_ZTRMVT (UPLO, N, T, LDT, X, INCX, Y, INCY, W, INCW,
Z, INCZ)
```

Input

UPLO	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.
N	Number of rows and columns in matrix T, $n > 0$.
T	REAL or COMPLEX array, dimension (LDT, N)—triangular matrix.
LDT	Leading dimension of array T. If **ldt** < 1 or **ldt** < **m**, an error flag is set and passed to the error handler.

	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., n, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_1 is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
	Y	REAL or COMPLEX array, minimum length (**N** - 1) x |**incy**| + 1.
	INCY	Increment for the array y. A vector y having component y_i, i = 1,..., n, is stored in an array **Y**() with increment argument **incy**. If **incy** > 0 then y_1 is stored in **Y**(1 + (i - 1) x **incy**). If **incy** < 0 then y_i is stored in **Y**(1 + (**N** - i) x |**incy**|). **incy** = 0 is an illegal value.
	INCW	Increment for the array w. A vector w having component w_i, i = 1,..., n, is stored in an array **W**() with increment argument **incw**. If **incw** > 0 then w_1 is stored in **W** (1 + (i - 1) x **incw**). If **incw** < 0 then w_i is stored in **W** (1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
	Z	REAL or COMPLEX array, minimum length (**N** - 1) x |**incz**| + 1.
	INCZ	Increment for the array y. A vector z having component z_i, i = 1,..., n, is stored in an array **Z**() with increment argument **incw**. If **incw** > 0 then z_1 is stored in **Z**(1 + (i - 1) x **incw**). If **incw** < 0 then z_i is stored in **Z**(1 + (**N** - i) x |**incw**|). **incw** = 0 is an illegal value.
Output	**X**	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1. Stores the result of the transpose matrix-vector multiply.
	W	REAL or COMPLEX array, minimum length (**N** - 1) x |**incw**| + 1. Stores the result of the matrix-vector multiply.

Name

F_STRSM/F_DTRSM/F_CTRSM/F_ZTRSM
Triangular solve

Purpose

F_xTRSM solves one of the following matrix equations:

$$B \leftarrow \alpha op(A^{-1})B$$

$$B \leftarrow \alpha Bop(A^{-1})$$

where α is a scalar, B is a general matrix, and A is a unit, or non-unit, upper or lower triangular matrix. op(A) denotes A, A^T, or A*.

The BLAS Standard {TR, TB, TP}SM Triangular Solve interface encompasses the legacy BLAS routine xTRSM with added functionality for triangular band and packed storage matrix-matrix multiplication.

Refer to "STRSM/DTRSM/CTRSM/ZTRSM" on page 250 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage

For these subprograms, you supply *A* in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If *A* has an unstored unit diagonal (see input argument **DIAG**), then the diagonal elements of the array also is not referenced.

Usage

```
INTEGER          DIAG, K, LDA, LDB, M, N, SIDE, TRANSA, UPLO
REAL*4           ALPHA, A( LDA, * ), B( LDB, * )
SUBROUTINE F_STRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A,
LDA, B, LDB)

INTEGER          DIAG, K, LDA, LDB, M, N, SIDE, TRANSA, UPLO
REAL*8           ALPHA, A( LDA, * ), B( LDB, * )
SUBROUTINE F_DTRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A,
LDA, B, LDB)

INTEGER          DIAG, K, LDA, LDB, M, N, SIDE, TRANSA, UPLO
COMPLEX*8        ALPHA, A( LDA, * ), B( LDB, * )
SUBROUTINE F_CTRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A,
LDA, B, LDB)

INTEGER          DIAG, K, LDA, LDB, M, N, SIDE, TRANSA, UPLO
COMPLEX*16       ALPHA, A( LDA, * ), B( LDB, * )
SUBROUTINE F_ZTRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A,
LDA, B, LDB)
```

Input

SIDE Specifies in which order the product of two matrices, A and B, are computed; $A \times B$ or $B \times A$. Use **BLAS_LEFT_SIDE** to specify A as the left matrix operand ($B \leftarrow \alpha op(A^{-1})B$), or **BLAS_RIGHT_SIDE** to specify A as the right matrix operand ($B \leftarrow \alpha Bop(A^{-1})$).

UPLO Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER**.

TRANS Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following constants:

BLAS_NO_TRANS

BLAS_TRANS

BLAS_CONJ_TRANS

DIAG Specifies whether the triangular matrix has unit-diagonal, that is $a_{i,i} = 1$, or not. Use either **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG**.

M Number of rows in matrix B, **m** ≥ 0. If **m** = 0, the subprograms do not reference **A** or **B**.

N Number of columns in matrix B, **n** ≥ 0. If **n** = 0, the subprograms do not reference **A** or **B**.

ALPHA The scalar ALPHA.

A Array whose upper or lower triangle, as specified by **uplo**, contains the unit, non-unit, upper or lower triangular matrix A. The matrix size is indicated by **SIDE**:

BLAS_LEFT_SIDE A is an m-by-m matrix

BLAS_RIGHT_SIDE A is an n-by-n matrix

LDA Leading dimension of array A.
For **SIDE** = **BLAS_LEFT_SIDE**, and **lda** < 1 or **lda** < **m**, an error flag is set and passed to the error handler.
For **SIDE** = **BLAS_RIGHT_SIDE**, and **lda** < 1 or **lda** < **n**, an error flag is set and passed to the error handler.

B Array containing the m by n matrix B. The representation of the matrix entry $b_{i,j}$ in B is denoted by B(i, j) for all (i, j) in the interval [0...m - 1] x [0...n - 1].

	LDB	Leading dimension of array B. For **SIDE** = **BLAS_LEFT_SIDE**, and **ldb** < 1 or **ldb** < **m**, an error flag is set and passed to the error handler. For **SIDE** = **BLAS_LEFT_SIDE**, and **ldb** < 1 or **ldb** < **m**, an error flag is set and passed to the error handler.
Output	**B**	The updated *m*-by-*n* matrix replaces the input.

Name F_STRSV/F_DTRSV/F_CTRSV/F_ZTRSV
Triangular solve

Purpose F_xTRSV solves one of the following equations:

$$x \leftarrow \alpha T^{-1} x$$

$$x \leftarrow \alpha T^{-T} x$$

$$x \leftarrow \alpha T^{-*} x$$

where x is a vector and the matrix T is a unit, non-unit, upper, or lower triangular matrix. T^{-T} is the inverse of the transpose of T, and T^{-*} is the inverse of the conjugate transpose of T.

Refer to "STRSV/DTRSV/CTRSV/ZTRSV" on page 254 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Matrix Storage For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **DIAG**), then the diagonal elements of the array also are not referenced.

Usage

```
INTEGER          DIAG, INCX, N, TRANS, UPLO
REAL*4           ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_STRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
REAL*8           ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_DTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
COMPLEX*8        ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_CTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA,  X,
INCX)

INTEGER          DIAG, INCX, N, TRANS, UPLO
COMPLEX*16       ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_ZTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)
```

Input	**UPLO**	Specifies whether a triangular matrix is upper or lower triangular. Use either **BLAS_UPPER** or **BLAS_LOWER.**
	TRANS	Specifies whether to apply the matrix (A), its transpose (A^T), or its conjugate transpose (A*). Use one of the following: **BLAS_NO_TRANS, BLAS_TRANS, or BLAS_CONJ_TRANS.**
	DIAG	Specifies whether the triangular matrix has unit-diagonal or not. Use one of the following: **BLAS_UNIT_DIAG** or **BLAS_NON_UNIT_DIAG.**
	N	Number of columns in matrix A, $n > 0$. If $n \leq 0$, the subprograms do not reference A or X.
	ALPHA	The scalar ALPHA.
	A	REAL or COMPLEX array, dimension (LDA, N).
	LDA	Leading dimension of array *A*. **lda** < 1 and **lda** < **n** are illegal values.
	X	REAL or COMPLEX array, minimum length (**N** - 1) x |**incx**| + 1.
	INCX	Increment for the array x. A vector x having component x_i, i = 1,..., *n*, is stored in an array **X**() with increment argument **incx**. If **incx** > 0 then x_i is stored in **X** (1 + (i - 1) x **incx**). If **incx** < 0 then x_i is stored in **X** (1 + (**N** - i) x |**incx**|). **incx** = 0 is an illegal value.
Output	**X**	The solution vector of the triangular system replaces the input.

4 Sparse Linear Equations

Overview

This chapter describes state-of-the-art software for the direct solution of sparse systems of linear equations with symmetric coefficient matrices or unsymmetric coefficient matrices that are structurally symmetric. Throughout this chapter, a system of linear equations is called "symmetric" if its coefficient matrix is both symmetric in structure and in value. "Symmetric in value" or simply "symmetric" means that $a(i,j) = a(j,i)$ for every i and j. The term "unsymmetric" means that the coefficient matrix is symmetric in structure, but not symmetric in value. "Symmetric in structure" or "structurally symmetric" means that if element (i,j) of the matrix is in the set of possible nonzeros, then element (j,i) also is in the set.

This package of subprograms provides efficient use of your computer's architecture in conjunction with powerful techniques for using the sparsity of a problem to reduce the cost of solution. Accuracy is assured through appropriate numerical techniques.

This chapter explains how to use the sparse linear equation subprograms to solve systems of sparse linear equations where the coefficient matrix is symmetric or unsymmetric but with symmetric structure. For a symmetric coefficient matrix, the solution is based on the LDL^T factorization with an option for pivoting. For coefficient matrices with symmetric structure but unsymmetric values, the solution is based on LU factorization without pivoting. Subprograms are provided to:

- Solve a single sparse linear system
- Estimate condition number of the coefficient matrix
- Efficiently solve multiple systems of equations

Chapter objectives

After you read this chapter you will:

- Understand what sparse systems are
- Understand how to use these subprograms to solve linear systems and to estimate condition numbers
- Understand issues in choosing an optimal method for a specific problem

This sparse matrix linear equation software makes it possible to call a single subprogram to solve a single system of sparse linear equations. However, this requires a particular format for the sparse matrix.

This package provides other approaches that provide a general interface to alternative representations of the sparse matrix, and also make available underlying capabilities for reducing the cost of solving multiple sparse systems. These optional approaches require the user to call a sequence of subprograms.

Associated documentation

The following documents provide supplemental material for this chapter:

Ashcraft, C.C. "A Vector Implementation of the Multifrontal Method for Large Sparse, Symmetric Positive Definite Linear Systems." *Boeing Computer Services Technical Report ETA-TR-51*. 1987.

Ashcraft, C.C. and R.G. Grimes. "The Influence of Relaxed Supernode Partitions on the Multifrontal Method." *ACM Transactions on Mathematical Software*. December, 1989. Vol. 15, No. 4. pp 291-309.

Ashcraft, C.C., R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon. "Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers." *The International Journal of Supercomputer Applications*. 1987. Vol. 1, No. 4. pp. 10-30.

Duff, I.S., A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Methods*. Oxford, England: Clarendon Press. 1986.

Duff, I.S. and J.K. Reid. "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations." *ACM Transactions of Mathematical Software*. September, 1983. Vol. 9, No. 3. pp. 302-325.

George, J.A. and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1981.

Heath, M.T., E. Ng, and B.W. Peyton. "Parallel Algorithms for Sparse Linear Systems." *SIAM Review*. September, 1991. Vol. 33, No. 3. pp. 420-460.

What you need to know to use these subprograms

Datatypes

For solving sparse systems of linear equations with matrix structure input by matrix, HP MLIB supports five datatypes:

- REAL*4
- REAL*8
- REAL*16
- COMPLEX*8
- COMPLEX*16

For solving sparse systems of linear equations with matrix structure input by elements, columns, and finite elements, HP MLIB supports REAL*8 precision.

For matrix structure input by matrix, HP MLIB can be used to solve multiple right-hand sides.

Sparsity and storage formats

Sparse matrices are matrices in which most of the entries are zero. The goal of sparse matrix software is to take maximum advantage of these zero entries to reduce storage and arithmetic. Storage is reduced by not storing zero entries; arithmetic is reduced by not performing operations on entries that are known to be zero.

It is easiest to see how to economize on storage. Suppose that an n-by-n matrix A has only nz nonzero entries. Then A could be specified completely by storing each of the nonzero values in an array of length nz that was accompanied by two integer arrays of length nz holding the corresponding row and column indices. Thus, $3nz$ storage suffices where n^2 storage is required for the corresponding dense matrix format.

Consider, for example, the following matrix:

11	0	13	14	0	0
0	22	23	0	25	0
31	32	33	0	35	0
41	0	0	44	45	0
0	52	53	54	55	0
0	0	0	0	0	66

This matrix could be represented in the format described above by three arrays, **irow**, **jcol**, and **mxvalu**, as shown in Table 4-1.

Figure 4-1 Row and Column Index Sparse Matrix Representation

irow	=	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	4	5	6
jcol	=	1	1	1	2	2	2	3	3	3	3	4	4	4	5	5	5	5	6
mxvalu	=	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	45	55	66

In this example, the matrix entries have been listed in order within each column, and the columns have been listed in order, although the representation does not require that much structure.

If the entries are required to be ordered by row and column, less storage is needed. In addition, symmetry in the matrix can be used by storing only the entries, for example, on or below the main diagonal. Other contexts, such as finite element analysis, can make even more concise representations of the locations of the nonzeros.

This package adopts a particular internal format that allows arbitrary matrices to be stored in $2nz+n+1$ storage locations, where nz represents the number of nonzeros in the matrix. In essence, the **jcol** array, which has repeated entries, is replaced by a shorter array that gives the index of the first element of each column in the **mxvalu** array and an indication of the number of elements in each column. These two pieces of information per matrix column can be represented in a single array **colstr** of length $n+1$ if the convention is adopted that the number of nonzeros in column j is given by **colstr**(j+1)–**colstr**(j) and if **colstr**(n+1) is set accordingly. When a matrix is symmetric, only its lower triangle is stored.

Both the full storage and lower triangular storage formats are known as the *column pointer, row index* representation. Table 4-2 illustrates the full storage format, and Table 4-3 shows the lower triangular representation.

Figure 4-2 **Column Pointer, Row Index Sparse Matrix Representation for the Full Matrix**

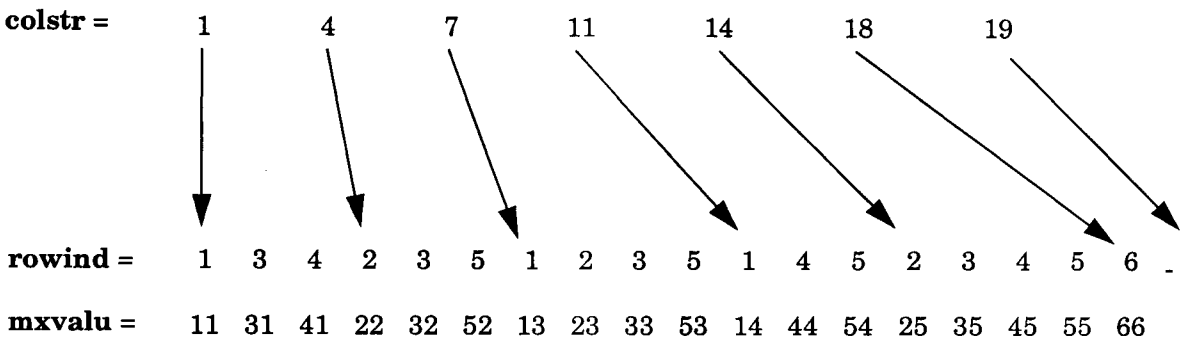
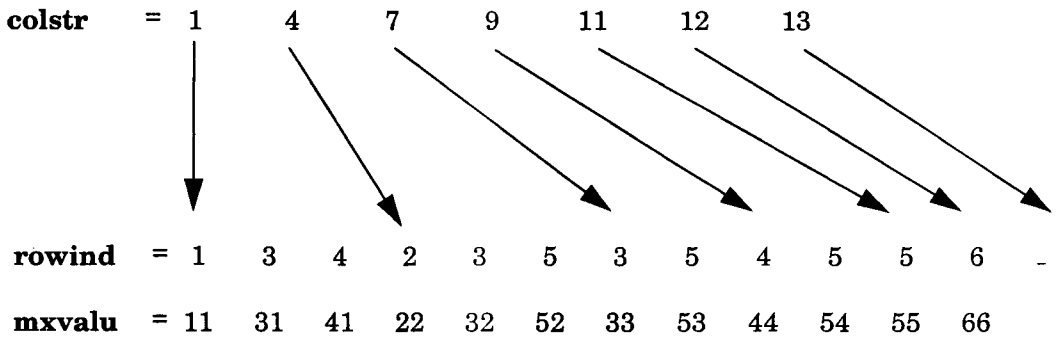


Figure 4-3 **Column Pointer, Row Index Sparse Matrix Representation for the Lower Triangle of the Matrix**



There are three ways of communicating the coefficient matrix to the package. One is a general form, which allows the user to store the matrix outside the package in whatever form is most convenient. The other two ways require that the user store the entire matrix in a form similar to the internal format or at least with all entries in each column contiguous in memory. Any of these three can be used. However, the general form carries additional overhead in computer time.

Direct versus iterative solution

This package is a *direct* linear equation solver. That is, it computes an explicit factorization of the matrix in a sparse analogy of the dense matrix subprograms DSYTRF/DSYTRS and DGETRF/DGETRS in LAPACK. There are applications where special, factorization-free algorithms can be used effectively. Algorithms appropriate for these cases *iterate* toward a solution, improving an approximate solution at each iteration.

Unfortunately, there is no generally effective iterative algorithm, only a collection of algorithms each effective for particular classes of problems. Used in the appropriate contexts, with only single or a few right-hand sides, and with only limited accuracy required of the solutions, iterative algorithms can be faster than direct methods. Used in the wrong contexts, iterative methods may become inordinately expensive and inaccurate. In contrast, the subprograms described here are designed to function well in general and, additionally, represent the algorithm of choice for many classes of problems.

Fill and reordering

When the matrix A is symmetric, this package computes its LDL^T factorization. When the matrix is unsymmetric, the package computes the LU factorization of A. It uses the factors to solve the system $Ax=b$. Here, L is a lower triangular matrix, D is diagonal or quasi-diagonal, and U is an upper triangular matrix. However, the sparsity of A is not sufficient to assure that L and U are sparse. Indeed, L and U have nonzeros wherever A has nonzeros, but also has nonzeros in other positions where A's entries are zero. These additional entries are known as *fill*. While fill is an intrinsic facet of the factorization process, the amount of fill is often controllable—if P is a matrix representing a permutation of the variables of the problem, the factorization of the matrix PAP^T, can often have significantly less fill than does the factorization of A, *provided that the permutation P is chosen appropriately*.

For example, it has been common in solving sparse systems of small order to choose permutations (to *reorder* the matrix) so that nonzero entries lie close to the main diagonal and then to use subprograms from LAPACK for banded matrices. Banded matrices are a special type of sparse matrix representation. This package is based on more general sparse matrix principles, often resulting in significantly less fill and correspondingly less work in computing the factorization. The principle is the same in both approaches in that the matrix must be reordered so that the factors are appropriately sparse. The reordering process precedes the actual factorization. The package solves the problem $(PAP^T)(Px) = (Pb)$. This transformation is invisible to the user.

The subprograms described in this chapter use a powerful heuristic, the *minimum degree algorithm*, for choosing the reordering. This algorithm is effective in general and is also efficient. In most cases, the cost of the reordering

process is recovered many times over by the reduction obtained in the cost of the factorization. In some applications, it is common to have several unknowns associated with each grid point of the solution domain. Sparse matrices derived from these applications tend to have some special properties in their structure. A technique known as compression can often improve the effectiveness of the minimum degree reordering. This package provides an option to turn on compression for these applications.

Optionally, users can turn on another ordering scheme based on the *multilevel nested dissection algorithm*, developed by George Karypis and Vipin Kumar from the University of Minnesota for the METIS package. The multilevel nested dissection algorithm has been seen to be effective for certain applications, such as matrices from linear programming (LP) problem and 3-D finite element problems using brick elements. Refer to the dsleop(3m) man page for details about selecting alternative reordering algorithms.

The numeric factorization algorithm processes columns with like structure simultaneously. This processing yields higher efficiency than older sparse matrix algorithms. The ordering is modified to collect columns of the factorization which have identical structure. One way to achieve higher computational rates at the cost of performing more numerical operations is to relax the ordering so that more columns can be collected together for simultaneous processing while allowing additional fill and hence more floating-point operations. This package provides a relaxation parameter, **maxzer**, which controls the amount of additional fill allowed for each set of collected columns. If **maxzer** = 0 no additional fill is allowed. Do not use a larger value without verifying that it actually improves performance.

Stability

This package is designed to solve three important classes of symmetric linear systems. One class comprises systems where the coefficient matrix is *positive definite*. Another class comprises the systems where the coefficient matrix is indefinite, but nonsingular. (Negative definite matrices can be treated by the positive definite subprograms by changing signs.) The third class comprises the systems where the coefficient matrix is not symmetric, but has symmetric structure.

The factorization $PAP^T = LDL^T$, with D diagonal, is always stable when A is positive definite. This has the particular effect that the reordering phase can choose any permutation P to maintain sparsity in the factor L. The factorization of an indefinite coefficient matrix is not always stable. However, in a manner closely related to LAPACK subprogram DSYTRF, a factorization can always be computed by allowing the matrix D to be a block diagonal matrix, where each block is either of size 1-by-1 or 2-by-2. It is also necessary to incorporate a permutation for *pivoting*, based on the numerical progress of the factorization, to create the proper 2-by-2 blocks that provides stability.

In the dense case, pivoting causes no difficulties. In the sparse case, pivoting can interfere with the permutation chosen to reduce fill. This conflict is dealt with in two ways in this sparse package. First, the effect of pivoting is localized in its modification of the sparsity reordering. Second, a pivoting tolerance is incorporated that allows the user to fine-tune the balance between sparsity and guarantees of stability. This is effected through a parameter, **pvttol**, to the factorization subprogram. **pvttol** is chosen in the range from 0 to 1. A choice of 0 gives a reordering based purely on sparsity. A choice of 1 gives the best guarantee of stability.

Although a purely sparse reordering often works for indefinite matrices, you should avoid using this option routinely for indefinite problems unless you have already verified its effectiveness on your particular class of problems.

If **pvttol** = 0, a faster algorithm is used than when **pvttol** > 0. Therefore, you should use **pvttol** = 0 if the matrix is known to be positive definite, or you know it is safe not to pivot for your particular class of problems.

When *A* is unsymmetric, the *LU* factorization is computed without pivoting. Therefore, it is the user's responsibility to validate the solution.

It is recommend that you compute matrix condition numbers to assure that sufficient accuracy can be obtained in the solution. Refer to the *LAPACK Users' Guide* for more details.

Global communications array

All of the subprograms in this package use dynamic memory allocation capabilities to free users from the often difficult issue of allocating storage for the factors of the matrix. This internal storage is invisible to the user. When sophisticated lower-level subprograms are used, knowledge of the internal storage is passed from subprogram to subprogram through a single communications array. This array, called **global** in each of the calling sequences, is a fixed-length double precision array of length 150. It must not be altered by the user because it represents the essential knowledge of the problem. Because it is the only identification for a problem, the user can handle multiple problems simultaneously by having multiple communications arrays with different names.

Error convention

Each subprogram has an error return flag, **ier**, as one of its arguments. A zero value returned in **ier** is the indication of successful processing. Fatal errors are signaled through negative values. An option allows you to print error messages in addition to returning an error flag. Refer to "Output controls" below.

Output controls

This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the *message level*. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and should generally be avoided. When **msglvl** = 1, some runtime statistics and any error messages are printed. When **msglvl** > 1, the complete set of runtime statistics are printed. If **msglvl** ≥ 3, various arrays whose length is on the order of the number of equations are printed. If **msglvl** ≥ 4, volumes of output are produced for debugging purposes.

It is recommend that you set **msglvl** = 1. The higher **msglvl** values (≥ 3) are intended for debugging purposes and generate copious amounts of printed output. Further, their use for this purpose may require some knowledge of the data structures being used by the package.

Paths of control

The key to using this package efficiently is to understand the possibilities for reusing work. As in solving dense linear systems, factored matrices can be reused as often as needed to solve additional systems. Thus, the subprogram DSLESL can be called repeatedly to solve additional systems with the same coefficient matrix after subprogram DSLEFA or DSLEFS has completed successfully. It is common to encounter sequences of sparse linear systems where the coefficient matrices change, but their sparsity structure, that is, the location of the nonzeros, remains fixed. In such cases, it is necessary to compute a factorization with DSLEFA for each coefficient matrix, but the work of choosing a reordering does not have to be repeated. The possible structures of reuse are illustrated in the following text, Figure 4-4, and Figure 4-5:

```
initialization
input matrix structure
reorder matrix
for m = 1 to number_of_structurally_identical_coefficient_matrices do
   input values of matrix entries
   factor matrix
   for r = 1 to
number_of_right_hand_sides_for_this_coefficient_matrix do
      solve
   endfor
endfor
```

Note that entering matrix values for a new coefficient matrix renders the previously computed factorization inaccessible. Similarly, entering matrix structure renders any previously reordered matrix structure inaccessible. However, the save and restart capabilities described in the utility subprograms

can be used to save multiple structures or factorizations, or to interrupt the package at any point.

Figure 4-4 Paths of Control—Sparse Linear Equations

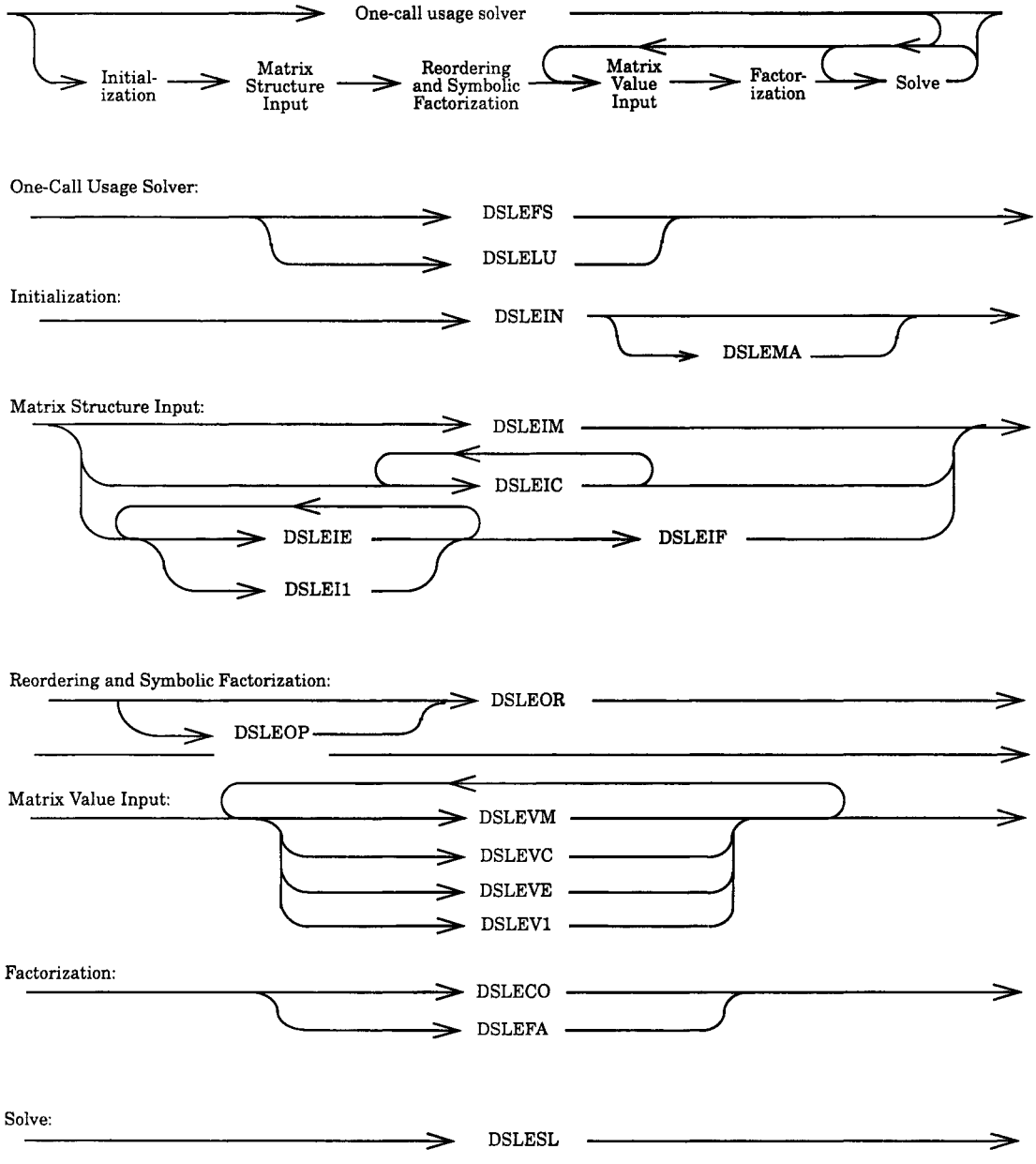
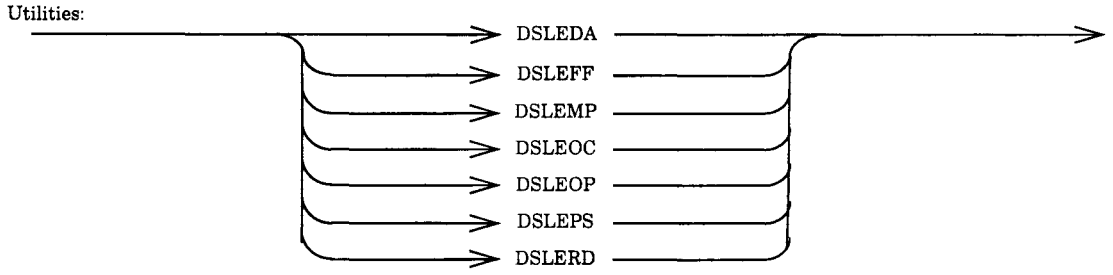


Figure 4-5 Paths of Control (continued)



Sample program

As illustrated in Figure 4-4 and Figure 4-5, there are many possible paths of control through this sparse matrix package. The following sample programs are provided to show one possible path through the package. It is intended to demonstrate that the package is not as difficult to use as Figure 4-4 and Figure 4-5 may imply.

In these examples, the row and column indices and the corresponding value for each nonzero entry of the matrix are assumed to be stored in the three arrays IROW, JCOL, and MXVALU. The right-hand-side vector is stored in the array RHS. These examples demonstrate the use of the subroutines for solving sparse systems of linear equations when subroutine DSLEFS cannot be used. The following code assumes that there are NNZERO nonzero entries in the matrix, which has NEQNS rows and columns. Values of NNZERO and NEQNS are set prior to this code segment.

Sample program for a symmetric matrix:

```
      INTEGER*4 I,J,K,NEQNS,IROW(NNZERO),JCOL(NNZERO),INRTIA(3),IER
      REAL*8    RCOND,GLOBAL(150),MXVALU(NNZERO),PVTTOL,VALUE
C
C     ... INITIALIZE THE SPARSE MATRIX PACKAGE
C
      CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C     ... INPUT THE MATRIX STRUCTURE (LOWER TRIANGULAR)
C

      DO 100 K = 1, NNZERO
          I = IROW(K)
          J = JCOL(K)
          CALL DSLEI1 (I,J,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  100 CONTINUE

      CALL DSLEIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C     ... REORDER THE MATRIX
C

      CALL DSLEOR (0,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     -
C     ... INPUT MATRIX VALUES (LOWER TRIANGULAR)
C     -

      DO 200 K = 1, NNZERO
          I     = IROW(K)
          J     = JCOL(K)
          VALUE = MXVALU(K)
          CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  200 CONTINUE

C     -
C     ... FACTOR THE MATRIX AND ESTIMATE ITS CONDITION NUMBER
C     -

      PVTTOL = 0.1D0
      CALL DSLECO (PVTTOL,RCOND,INRTIA,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     -
C     ... SOLVE FOR A GIVEN RIGHT HAND SIDE
C     -

      CALL DSLESL (1,RHS,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
```

```
C
C     ... USE THE SOLUTION STORED IN ARRAY RHS
C
      .
      .
      .
C
C     ... ERROR TRAP
C

 8000  ..........
```

Sample program for a nonsymmetric matrix:

```
      INTEGER*4 I,J,K,NEQNS,IROW(NNZERO),JCOL(NNZERO),INRTIA(3),IER
      REAL*8    GLOBAL(150),MXVALU(NNZERO),PVTTOL,VALUE
C
C     ... INITIALIZE THE SPARSE MATRIX PACKAGE
C
      CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

      CALL DSLEMA ("SU", GLOBAL, IER)
      IF (IER .NE. 0) GO TO 8000
C
C     ... INPUT THE MATRIX STRUCTURE (FULL MATRIX)
C

      DO 100 K = 1, NNZERO
          I = IROW(K)
          J = JCOL(K)
          CALL DSLEI1 (I,J,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  100 CONTINUE

      CALL DSLEIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C     ... REORDER THE MATRIX
C

      CALL DSLEOR (0,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     -
C     ... INPUT MATRIX VALUES (FULL MATRIX)
C     -

      DO 200 K = 1, NNZERO
          I     = IROW(K)
          J     = JCOL(K)
          VALUE = MXVALU(K)
          CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  200 CONTINUE
```

Sample program

```
C     -
C     ... FACTOR THE MATRIX
C     -

      PVTTOL = 0.0D0
      CALL DSLEFA (PVTTOL,INRTIA,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     -
C     ... SOLVE FOR A GIVEN RIGHT HAND SIDE
C     -

      CALL DSLESL (1,RHS,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C     ... USE THE SOLUTION STORED IN ARRAY RHS
C
      .
      .
      .
C
C     ... ERROR TRAP
C

 8000  ..........
```

Subprograms for sparse linear equations

The following sections describe subprograms included in VECLIB for the direct solution of sparse symmetric or nonsymmetric but structurally symmetric linear equations.

Name

DSLEFS
One-call usage

Purpose

This subprogram provides one-call interface to the sparse linear equation solution package for a symmetric matrix based on the LDL^T factorization. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package.

Usage

INTEGER*4 **neqns, maxzer, msglvl, output, colstr(neqns+1), rowind(nnzero), nrhs, ldrhs, inrtia(3), ier**

REAL*8 **pvttol, value(nnzero), rhs(ldrhs, nrhs), rcond, global(150)**

CALL DSLEFS(neqns, maxzer, pvttol, msglvl, output, colstr, rowind, value, nrhs, rhs, ldrhs, rcond, inrtia, global, ier)

Input

neqns Number of equations; **neqns** > 0.

maxzer Supernodal relaxation parameter; **maxzer** ≥ 0. If **maxzer** > 0, the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization. If **maxzer** = 0, no additional fill is allowed. If many solution vectors are to be computed for each factorization, it is suggested that **maxzer** = 0 be used. If only a few solution vectors are to be computed **maxzer** = 10 could be tried but **maxzer** = 0 is perfectly acceptable. Using values greater than 50 probably increases the factorization time.

pvttol Pivoting tolerance; 0 ≤ **pvttol** ≤ 1. The value 0 gives a reordering that minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite.

	msglvl	Message level for printable output. Options: **msglvl** ≤ 0 Suppress all output. **msglvl** = 1 Error messages and summary statistics. **msglvl** = 2 More complete statistics. **msglvl** = 3 First stage of debugging output. **msglvl** = 4 Complete debugging output.
	output	Fortran logical unit number to which all printable output is written.
	colstr	**colstr**(*j*) gives the index in **rowind** of the first nonzero in the lower triangular part of column *j* of the matrix. All of the nonzeros for column *j* are found, in ascending order, in **rowind**(**colstr**(*j*)), **rowind**(**colstr**(*j*)+1), ..., **rowind**(**colstr**(*j*+1)–1). **colstr**(**neqns**+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the lower triangular part of the matrix.
	rowind	List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix *A*.
	value	List of values corresponding in position to the indices in **rowind**.
	nrhs	Number of right-hand sides.
	ldrhs	The leading dimension of array **rhs** as declared in the calling program unit, with **ldrhs** ≥ **neqns**.
Updated	**rhs**	On input, **rhs** contains the **nrhs** right-hand sides. On output, **rhs** has been overwritten with the computed solutions.
Output	**rcond**	Estimate of the reciprocal of the 1-norm condition number.
	inrtia	Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier	Status response:	
	ier = 0	Normal return.
	ier = –101	Error in dynamic storage allocation during matrix structure input.
	ier = –102	**neqns** ≤ 0.
	ier = –106	Unacceptable matrix structure.
	ier = –201	Error in dynamic storage allocation during ordering.
	ier = –301	Error in dynamic storage allocation during symbolic factorization.
	ier = –302	Internal error.
	ier = –401	Error in dynamic storage allocation during value input.
	ier = –501	Error in dynamic storage allocation during factorization.
	ier = –502	Error in dynamic storage allocation during factorization.
	ier = –503	Exactly zero pivot encountered during factorization; matrix is singular.
	ier = –602	**nrhs** ≤ 0.
	ier = –603	**ldrhs** < **neqns**.

Notes Calling DSLEFS is equivalent to calling DSLEIN, DSLEIM, DSLEOR, DSLEVM, DSLECO, and DSLESL in sequence. You can follow the call to DSLEFS with other calls to subprograms in the package to solve additional right-hand sides, to input new matrix values, to input a new matrix structure, or to perform utility functions.

Example Solve the small (order 6) sparse system of linear equations where the matrix A and right-hand side b are

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix} \qquad b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

A supernode relaxation factor of 0 (no extra fill allowed) and a pivoting tolerance of 0.1 are used. Printed output is to Fortran logical unit 6. Only printed error messages and some runtime statistics are printed.

```
INTEGER*4 COLSTR(7),ROWIND(13),INRTIA(3),IER
REAL*8    VALUES(13),RHS(6),RCOND,GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 4
COLSTR(3) = 7
COLSTR(4) = 9
COLSTR(5) = 12
COLSTR(6) = 13
COLSTR(7) = 14

ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 4
ROWIND(4) = 2
ROWIND(5) = 3
ROWIND(6) = 5
ROWIND(7) = 3
ROWIND(8) = 5
ROWIND(9) = 4
ROWIND(10) = 5
ROWIND(11) = 6
ROWIND(12) = 5
ROWIND(13) = 6

VALUES(1) = 4.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 4.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0
VALUES(7) = 4.0D0
VALUES(8) = 1.0D0
VALUES(9) = 4.0D0
VALUES(10) = 1.0D0
VALUES(11) = 1.0D0
VALUES(12) = 4.0D0
VALUES(13) = 1.0D0

RHS(1) = 1.0D0
RHS(2) = 0.0D0
RHS(3) = 1.0D0
RHS(4) = 1.0D0
RHS(5) = 0.0D0
RHS(6) = 0.0D0

CALL DSLEFS (6,0,0.1D0,1,6,COLSTR,ROWIND,VALUE,
             1,RHS,6,RCOND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLELU
One-call usage

Purpose

This subprogram provides a one-call interface to the sparse linear equation solution package for nonsymmetric matrices based on the *LU* factorization. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the sparse linear equation package.

Usage

INTEGER*4 **neqns, maxzer, msglvl, output, colstr(neqns+1), rowind(nnzero), nrhs, ldrhs, inrtia(3), ier**

REAL*8 **pvttol, value(nnzero), rhs(ldrhs, nrhs), rcond, global(150)**

CALL DSLELU(neqns, maxzer, pvttol, msglvl, output, colstr, rowind, value, nrhs, rhs, ldrhs, rcond, inrtia, global, ier)

Input

neqns Number of equations; **neqns** > 0.

maxzer Supernodal relaxation parameter; **maxzer** ≥ 0. If **maxzer** > 0, the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization. If **maxzer** = 0, no additional fill is allowed. If many solution vectors are to be computed for each factorization, it is suggested that **maxzer** = 0 be used. If only a few solution vectors are to be computed **maxzer** = 10 could be tried but **maxzer** = 0 is perfectly acceptable. Using values greater than 50 probably increases the factorization time.

pvttol Not used in this case.

msglvl Message level for printable output. Options:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output Fortran logical unit number to which all printable output is written.

nrhs Number of right-hand sides.

ldrhs The leading dimension of array **rhs** as declared in the calling program unit, with **ldrhs** ≥ **neqns**.

	colstr	**colstr**(j) gives the index in **rowind** of the first nonzero in column j of the matrix. All of the nonzeros for column j are found, in ascending order, in **rowind**(**colstr**(j)), **rowind**(**colstr**(j)+1), ..., **rowind**(**colstr**(j+1)–1). **colstr**(**neqns**+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the matrix.
	rowind	List of row indices for all nonzeros, in ascending order within each column, in the matrix A.
	value	List of values corresponding in position to the indices in **rowind**.
Updated	**rhs**	On input, **rhs** contains the **nrhs** right-hand sides. On output, **rhs** has been overwritten with the computed solutions.
Output	**ier**	Status response:

ier = 0	Normal return.
ier = –101	Error in dynamic storage allocation during matrix structure input.
ier = –102	**neqns** ≤ 0.
ier = –106	Unacceptable matrix structure.
ier = –201	Error in dynamic storage allocation during ordering.
ier = –301	Error in dynamic storage allocation during symbolic factorization.
ier = –302	Internal error.
ier = –401	Error in dynamic storage allocation during value input.
ier = –501	Error in dynamic storage allocation during factorization.
ier = –502	Error in dynamic storage allocation during factorization.
ier = –503	Exactly zero pivot encountered during factorization; matrix is singular.
ier = –602	**nrhs** ≤ 0.
ier = –603	**ldrhs** < **neqns**.

rcond	Unused as output. Reserved for future use.
inrtia	Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.
global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Notes Calling DSLELU is equivalent to calling DSLEIN, DSLEMA, DSLEIM, DSLEOP, DSLEOR, DSLEVM, DSLEFA, and DSLESL in sequence. You may follow the call to DSLELU with other calls to subprograms in the package to solve additional right-hand sides, to input new matrix values, to input a new matrix structure, or to perform utility functions.

Example Solve the small (order 6) sparse system of linear equations where the matrix A and right-hand side b are

$$A = \begin{bmatrix} 4.0 & 0.0 & -1.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & -1.0 & 0.0 & -1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & -1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & -1.0 & -1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{bmatrix} \quad b = \begin{bmatrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

A supernode relaxation factor of 0 (no extra fill allowed) and a pivoting tolerance of 0.1 is used. Printed output is to Fortran logical unit 6.

Only printed error messages and some runtime statistics are printed.

```
INTEGER*4 COLSTR(7),ROWIND(13),INRTIA(3),IER
REAL*8    VALUES(13),RHS(6),RCOND,GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 4
COLSTR(3) = 7
COLSTR(4) = 11
COLSTR(5) = 15
COLSTR(6) = 19
COLSTR(7) = 21

ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 4
ROWIND(4) = 2
ROWIND(5) = 3
ROWIND(6) = 5
ROWIND(7) = 1
```

```
ROWIND(8) = 2
ROWIND(9) = 3
ROWIND(10) = 5
ROWIND(11) = 1
ROWIND(12) = 4
ROWIND(13) = 5
ROWIND(14) = 6
ROWIND(15) = 2
ROWIND(16) = 3
ROWIND(17) = 4
ROWIND(18) = 5
ROWIND(19) = 4
ROWIND(20) = 6

VALUES(1) = 4.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 4.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0
VALUES(7) = -1.0D0
VALUES(8) = -1.0D0
VALUES(9) = 4.0D0
VALUES(10) = 1.0D0
VALUES(11) = -1.0D0
VALUES(12) = 4.0D0
VALUES(13) = 1.0D0
VALUES(14) = 1.0D0
VALUES(15) = -1.0D0
VALUES(16) = -1.0D0
VALUES(17) = -1.0D0
VALUES(18) = 4.0D0
VALUES(19) = -1.0D0
VALUES(20) = 1.0D0

CALL DSLELU (6,0,0.1D0,1,6,COLSTR,ROWIND,VALUE,
             1,RHS,6,RCOND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEIN
Initialize sparse linear equations

Purpose

This subprogram provides the necessary information to begin processing with the sparse linear equation solution package. For a nonsymmetric matrix (*LU* factorization), this call must be followed by a call to DSLEMA.

Usage

INTEGER*4 **neqns, msglvl, output, ier**
REAL*8 **global(150)**
CALL DSLEIN(neqns, msglvl, output, global, ier)

Input

neqns Order of matrix (number of degrees of freedom); **neqns** > 0.

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output Fortran logical unit number to which all printable output is written.

Output

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier Status response:

ier = 0	Normal return.
ier = –100	Too many systems initialized.
ier = –101	Error in dynamic storage allocation.
ier = –102	**neqns** ≤ 0.

Example

Prepare to solve a system of equations of order 10,000. Obtain error messages and standard minimal output on Fortran logical unit 6.

```
INTEGER*4 NEQNS,IER
REAL*8    GLOBAL(150)
NEQNS = 10000
CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSLEMA
Specify coefficient matrix type of sparse linear equations

Purpose This subprogram specifies whether the coefficient matrix is symmetric, or unsymmetric but with symmetric structure. A call to DSLEMA is optional for symmetric case. For the unsymmetric case, DSLEMA must be called immediately after DSLEIN.

Usage

```
CHARACTER*2    type
INTEGER*4      ier
REAL*8         global(150)
CALL DSLEMA(type, global, ier)
```

Input

type Type of matrix:

type = 'SS' or 'ss'	Symmetric matrix.
type = 'SU' or 'su'	Unsymmetric matrix with symmetric structure.

Output

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier Status response:

ier = 0	Normal return.
ier = –110	Incorrect processing path; DSLEIN not called.

Name

DSLEI1
Matrix structure input by single entry

Purpose

This subprogram adds a single entry in the (**irow, jcol**) position to the set of known nonzeros for the sparse matrix.

Usage

```
INTEGER*4      irow, jcol, ier
REAL*8         global(150)
CALL DSLEI1(irow, jcol, global, ier)
```

Input

irow — Row index of the nonzero entry. For a symmetric matrix (*LDL'* factorization), **jcol** ≤ **irow** ≤ **neqns**, where **neqns** is the number of equations specified in the call to DSLEIN. Input of diagonal entries is optional.

jcol — Column index of the nonzero entry; 1 ≤ **jcol** ≤ **neqns**, where **neqns** is the number of equations specified in the call to DSLEIN.

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier — Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –104	Illegal value for **jcol**.
ier = –105	Illegal value for **irow**.

Notes

Calls to DSLEI1 and DSLEIE can be intermixed. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE are used.

Example

Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix.

```
INTEGER*4 I,J,IER
REAL*8    GLOBAL(150)
I = 3035
J = 1024
CALL DSLEI1 (I,J,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEIC
Matrix structure input by column

Purpose

This subprogram adds a list of indices in a single column to the set of known nonzeros for the sparse matrix.

Usage

```
INTEGER*4    jcol, nzcol, jrowin(nzcol), ier
REAL*8       global(150)
CALL DSLEIC(jcol, nzcol, jrowin, global, ier)
```

Input

jcol — Column index; 1 ≤ **jcol** ≤ **neqns**. Columns must be presented in order from 1 to **neqns**, except that columns with no entries can be skipped.

nzcol — Number of nonzeros in column **jcol** of the matrix; **nzcol** ≥ 0.

jrowin — For a symmetric matrix, (*LDL'* factorization), a list of row indices for all nonzeros, in ascending order, in the lower triangular part of column **jcol** of the matrix; **jcol** ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) ≤ **neqns**.

For a nonsymmetric matrix (*LU* factorization), a list of row indices for all nonzeros, in ascending order, in column **jcol** of the matrix; 1 ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) ≤ **neqns**

Input of diagonal entries is optional

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier — Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –103	Illegal value for **nzcol**.
ier = –104	Illegal value for **jcol**, or out of order.
ier = –106	Illegal value for at least one entry in **jrowin** or entries out of order.

Notes The entire matrix structure must be input with DSLEIC if it is used. Its use is not compatible with DSELE1, DSLEIE, or DSLEIM.

If the matrix entries are available by column, using DSLEIC is more efficient than using DSLEI1.

Example 1 Column 4519 of a symmetric matrix has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002 and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix. Columns 1 to 4518 already have been input to the package using DSLEIC.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)

J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004

CALL DSLEIC (J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Example 2 Column 4519 of a nonsymmetric matrix has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002 and 6004. Add all seven entries to the list of nonzeros in the matrix. Columns 1 to 4518 already have been input to the package using DSLEIC.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)

J = 4519
NZCOL = 7
JROWIN(1) = 1
JROWIN(2) = 2735
JROWIN(3) = 4519
JROWIN(4) = 4520
JROWIN(5) = 4521
JROWIN(6) = 6000
JROWIN(7) = 6002
JROWIN(8) = 6004

CALL DSLEIC (J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEIE
Matrix structure input by finite element

Purpose

This subprogram adds indices to the set of known nonzeros in the lower triangle of the sparse matrix corresponding to a finite element or clique.

Usage

```
INTEGER*4      nnode, nodlst(nnode), ier
REAL*8         global(150)
CALL DSLEIE(nnode, nodlst, global, ier)
```

Input

nnode	Number of nodes in the finite element or clique.
nodlst	List of nodes. All pairs (**nodlst**(*i*),**nodlst**(*j*)) in the lower triangle are added to the sparsity structure of the matrix.

Updated

nodlst	The order of the values in **nodlst** may be changed.
global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –107	Illegal value for **nnode**.
ier = –108	Illegal value for at least one entry in **nodlst**.

Notes

Calls to DSLEIE can be intermixed with calls to DSLEI1. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE is used.

Example Rows and columns 345, 346, 347 and 989 form a small dense submatrix of A. Add the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989

CALL DSLEIE (NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEIM
Matrix structure input by matrix

Purpose

This subprogram specifies the locations of nonzeros of the sparse matrix all at once. For a symmetric matrix (*LDL'* factorization), only the locations of the nonzeros in the lower triangle are specified. For a nonsymmetric matrix (*LU* factorization), the locations of all nonzeros in the sparse matrix are specified.

Usage

```
INTEGER*4    neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier
REAL*8       global(150)
CALL DSLEIM(colstr, rowind, global, ier)
```

Input

colstr

For a symmetric matrix, **colstr**(*j*) gives the index in **rowind** of the first nonzero in the lower triangular part of column *j* of the matrix.

For a nonsymmetric matrix, **colstr**(*j*) gives the index in **rowind** of the first nonzero in column *j* of the matrix.

The nonzeros for column *j* are found, in ascending order, in **rowind(colstr**(*j*)), **rowind(colstr**(*j*)+1), ..., **rowind(colstr**(*j*+1)–1).

colstr(neqns+1) must be set to one greater than the total number of nonzeros, **nnzero** listed in **rowind**.

rowind

For a symmetric matrix (*LDL'* factorization), a column-by-column list of row indices for all nonzeros in the lower triangle of the matrix, in ascending order within each column.

For a nonsymmetric matrix (*LU* factorizations), a column-by-column list of row indices for all nonzeros in the matrix, in ascending order within each column.

Input of diagonal entries is optional.

Updated

global

Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier

Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.

ier = –101	Error in dynamic storage allocation.
ier = –106	Illegal value for at least one entry in **colstr** (not increasing or negative) or invalid entries in **rowind** (entries out of order within a column or out of the lower triangle).

Notes This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure must be input with DSLEIM if it is used. Its use is not compatible with DSLEI1, DSLEIE, or DSLEIC.

Example 1 Input a small (order 6) symmetric sparse matrix with this structure:

```
x  0  x  x  0  0
0  x  x  0  x  0
x  x  x  0  x  0
x  0  0  x  x  0
0  x  x  x  x  0
0  0  0  0  0  x
```

```
INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

CALL DSLEIM (COLSTR,ROWIND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Example 2 Input a small (order 6) nonsymmetric symmetric sparse matrix with this structure:

```
x  0  x  x  0  0
0  x  x  0  x  0
x  x  x  0  x  0
x  0  0  x  x  0
0  x  x  x  x  0
0  0  0  0  0  x
```

```
INTEGER*4 COLSTR(7),ROWIND(12),IER
REAL*8    GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 8
COLSTR(5) = 10
COLSTR(6) = 13
COLSTR(7) = 13

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 1
ROWIND(6) = 2
ROWIND(7) = 5
ROWIND(8) = 1
ROWIND(9) = 5
ROWIND(10) = 2
ROWIND(11) = 3
ROWIND(12) = 4

CALL DSLEIM (COLSTR,ROWIND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEIF
End of matrix structure input

Purpose

This subprogram specifies the end of structure input for the matrix. DSELIF is used only if routines DSLEI1 and/or DSLEIE were the mechanism by which the structure was input.

Usage

```
INTEGER*4       ier
REAL*8          global(150)
CALL DSLEIF(global, ier)
```

Updated

global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier	Status response:	
	ier = 0	Normal return.
	ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
	ier = –101	Error in dynamic storage allocation.

Example

The nonzero structure of a pair of finite element matrices was passed to the package using repeated calls to subroutine DSLEIE. Signal that no more nonzeros will be added to the matrix.

```
INTEGER*4  IER
REAL*8     GLOBAL(150)
CALL DSLEIF (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSLEOR
Reordering and symbolic factorization

Purpose This subprogram reorders the matrix whose pattern of nonzeros has been input to the package to obtain an efficient sparse factorization. It then constructs data structures that represent the factorization. Optionally, different ordering heuristics can be selected by a call to DSLEOP before the call to DSLEOR.

Usage

```
INTEGER*4      maxzer, ier
REAL*8         global(150)
CALL DSLEOR(maxzer, global, ier)
```

Input **maxzer** Supernodal relaxation parameter; **maxzer** ≥ 0.

If **maxzer** > 0, the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization.

If **maxzer** = 0 no additional fill is allowed. Performance studies indicate that a value of 20 to 25 optimize the factorization execution time with a reduction of 5 to 10%. Because of the additional fill, the solution execution time may increase by 1 to 2%.

If many solution vectors are to be computed for each factorization, it is suggested that **maxzer** = 0 be used.

If only a few solution vectors are to be computed **maxzer** = 25 could be tried but **maxzer** = 0 is perfectly acceptable. Using values greater than 100 probably increase the factorization time.

Updated **global** Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –200	Incorrect processing path; structure input not completed or value input subroutines already called.
ier = –201	Error in dynamic storage allocation.

Example The sparse matrix has been communicated to the page using DSLEIN and DSLEI1, DSLEIE, and DSLEIF, and DSLEIC or DSLEIM. The next step is obtaining good reordering and performing symbolic factorization for the numeric factorization phase. The value of MAXZER = 0 is used so that no extra fill occurs in the factorization.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSLEOR (0,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEV1
Matrix value input by single entry

Purpose

This subprogram adds to the value of the entry in the (**irow, jcol**) position of the sparse matrix.

Usage

```
INTEGER*4      irow, jcol, ier
REAL*8         value, global(150)
CALL DSLEV1(irow, jcol, value, global, ier)
```

Input

irow	Row index of the nonzero entry. For a symmetric matrix (LDL' factoriation), **jcol** ≤ **irow** ≤ **neqns**.
jcol	Column index of the nonzero entry, 1 ≤ **jcol** ≤ **neqns**.
value	Numeric value that is added to any previous values input for this location.

Updated

global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	Subscript pair (**irow, jcol**) was not specified in structure input. No room for value.

Notes

Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed.

Example

Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix.

```
INTEGER*4 I,J,IER
REAL*8    VALUE,GLOBAL(150)
I = 3035
J = 1024
VALUE = 4.523D-5
CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEVC
Matrix value input by column

Purpose

This subprogram adds to the values of a list of nonzero entries in a single column of the sparse matrix.

Usage

```
INTEGER*4      jcol, nzcol, jrowin(nzcol), ier
REAL*8         values(nzcol), global(150)
CALL DSLEVC(jcol, nzcol, jrowin, values, global, ier)
```

Input

jcol — Column index; $1 \le$ **jcol** $\le$ **neqns**.

nzcol — Number of nonzeros in the list of nonzeros for column **jcol** of the matrix; **nzcol** ≥ 0.

jrowin — For a symmetric matrix (*LDL'* factorization), a list of row indices for nonzeros, in ascending order, in the lower triangular part of column **jcol** of the matrix; **jcol** $\le$ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) $\le$ **neqns**.

For a nonsymmetric matrix (*LU* factorization), a list of row indices for nonzeros, in ascending order, in column **jcol** of the matrix; $1 \le$ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) $\le$ **neqns**. Exceptionally, if present, the diagonal entry can either be the first element of **jrowin** or ascendingly ordered as described above.

values — List of values corresponding to the positions specified by **jcol** and **jrowin**.

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier — Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	At least one subscript pair (**jrowin**(k),**jcol**) was not specified in structure input. No room for value.

Notes Calls to DSLEV1, DSLEVC, DSLEVE and DSLEVM can be intermixed. It is not necessary that all entries in the column **jcol** be included in **jrowin**. If the matrix entries are available by column, using DSLEVC is more efficient than using DSLEV1.

Example 1 Column 4519 of a symmetric matrix has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 6
JROWIN(1) = 4519
JROWIN(2) = 4520
JROWIN(3) = 4521
JROWIN(4) = 6000
JROWIN(5) = 6002
JROWIN(6) = 6004

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0

CALL DSLEVC (J,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Example 2 Column 4519 of a nonsymmetric matrix has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 8
JROWIN(1) = 1
JROWIN(2) = 2735
JROWIN(3) = 4519
JROWIN(4) = 4520
JROWIN(5) = 4521
JROWIN(6) = 6000
JROWIN(7) = 6002
JROWIN(8) = 6004

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0
VALUES(7) = 1.0D0
VALUES(8) = 1.0D0

CALL DSLEVC (J,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEVE
Matrix value input by finite element

Purpose

This subprogram adds to the values of a set of nonzero entries in the lower triangle of the sparse matrix corresponding to a finite element or clique.

Usage

```
INTEGER*4      nnode, ldelmx, nodlst(nnode), ier
REAL*8         elmmtx(ldelmx, nnode), global(150)
CALL DSLEVE(nnode, nodlst, elmmtx, ldelmx, global, ier)
```

Input

nnode Number of nodes in the finite element or clique.

nodlst List of nodes in element or clique. Values for all pairs (**nodlst**(i),**nodlst**(j)) are added to the values of the matrix.

elmmtx Array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of **elmmtx** is referenced. The value in **elmmtx**(k,l) is added to the value in position (**nodlst**(k),**nodlst**(l)) in the sparse matrix.

ldelmx The leading dimension of array **elmmtx** as declared in the calling program unit, with **ldelmx** ≥ **nnode**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	At least one subscript pair (**nodlst**(k),**nodlst**(l)) was not specified in structure input. No room for value.

Notes

Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed. Using DSLEVE is more efficient in finite element contexts, where the matrix is created as the sum of elemental matrices, if the user has stored the elemental matrices rather than the assembled matrix.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8    ELMMTX(10,10),GLOBAL(150)
      NNODE = 4
      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989
      DO 200 K = 1, NNODE
          DO 100 L = K, NNODE
              ELMMTX(K,L) = 1.0D0
100       CONTINUE
200   CONTINUE
      CALL DSLEVE (NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
          handle error condition
      ENDIF
```

Name DSLEVM
Matrix value Input by matrix

Purpose This subprogram adds to the values of many or all of the nonzero entries in the sparse matrix.

Usage

```
INTEGER*4    neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier
REAL*8       values(nnzero), global(150)
CALL DSLEVM(colstr, rowind, values, global, ier)
```

Input

colstr For a symmetric matrix, **colstr**(*j*) gives the index in **rowind** of the first nonzero in the lower triangular part of column *j* of the matrix.

For a nonsymmetric matrix, **colstr**(*j*) gives the index in **rowind** of the first nonzero in column *j* of the matrix.

The nonzeros for column *j* are found, in ascending order, in **rowind**(**colstr**(*j*)), **rowind**(**colstr**(*j*)+1), ..., **rowind**(**colstr**(*j*+1)–1).

colstr(**neqns**+1) must be set to one greater than the total number of nonzeros, **nnzero**, listed in **rowind**.

rowind For a symmetric matrix (*LDL'* factorization), a column-by-column list of row indices for many or all nonzeros in the lower triangle of the matrix, in ascending order within each column.

For a nonsymmetric matrix (*LU* factorization), a column-by-column list of row indices for many or all nonzeros in the matrix, in ascending order within each column.

Diagonal entries may be present as well.

values List of values corresponding in position to the indices in **rowind**. These values are added to any values already present in the matrix.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	At least one subscript pair was not specified in structure input. No room for value.

Notes This is the most efficient mechanism for specifying the nonzero values. Normally, DSLEVM is used in conjunction with DSLEIM. Additional entries or modifications can be entered with DSLEV1, DSLEVC, or DSLEVE.

Example 1 Input the values for the following small (order 6) symmetric sparse matrix problem.

4.0	0.0	1.0	1.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	0.0	1.0	0.0
1.0	0.0	0.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	4.0

Use DSLEVM with the same structure used in "Example 1" of the documentation for DSLEIM, and DSLEV1 to input this matrix.

```
INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    VALUES(6),DIAGVL,GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
```

```
      VALUES(5) = 1.0D0
      VALUES(6) = 1.0D0

      CALL DSLEVM (COLSTR,ROWIND,VALUES,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
          handle error condition
      ENDIF

      DIAGVL = 4.0
      DO 100 I = 1, NEQNS
          CALL DSLEV1 (I,I,DIAGVL,GLOBAL,IER)
          IF ( IER .NE. 0 ) THEN
              handle error condition
          ENDIF
  100 CONTINUE
```

This matrix could also have been input strictly with DSLEVM by expanding COLSTR, ROWIND, and VALUES to include the diagonal. This action would have avoided the calls to DSLEV1. Including the diagonal entries in COLSTR and ROWIND is also acceptable to DSLEIM.

Example 2 Input the values for the following small (order 6) nonsymmetric sparse matrix problem.

4.0	0.0	−1.0	−1.0	0.0	0.0
0.0	4.0	−1.0	0.0	−1.0	0.0
1.0	1.0	4.0	0.0	−1.0	0.0
1.0	0.0	0.0	4.0	−1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	4.0

Use DSLEVM with the same structure used in "Example 2" of the documentation for DSLEIM, and DSLEV1 to input this matrix.

```
      INTEGER*4 COLSTR(7),ROWIND(12),IER
      REAL*8    VALUES(12),DIAGVL,GLOBAL(150)

      COLSTR(1) = 1
      COLSTR(2) = 3
      COLSTR(3) = 5
      COLSTR(4) = 8
      COLSTR(5) = 10
      COLSTR(6) = 12
      COLSTR(7) = 12

      ROWIND(1) = 3
      ROWIND(2) = 4
      ROWIND(3) = 3
      ROWIND(4) = 5
      ROWIND(5) = 1
      ROWIND(6) = 2
      ROWIND(7) = 5
```

```
      ROWIND(8) = 1
      ROWIND(9) = 5
      ROWIND(10) = 2
      ROWIND(11) = 3
      ROWIND(12) = 4

      VALUES(1) = 1.0D0
      VALUES(2) = 1.0D0
      VALUES(3) = 1.0D0
      VALUES(4) = 1.0D0
      VALUES(5) = -1.0D0
      VALUES(6) = -1.0D0
      VALUES(7) = 1.0D0
      VALUES(8) = -1.0D0
      VALUES(9) = 1.0D0
      VALUES(10) = -1.0D0
      VALUES(11) = -1.0D0
      VALUES(12) = -1.0D0

      CALL DSLEVM (COLSTR,ROWIND,VALUES,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
          handle error condition
      ENDIF

      DIAGVL = 4.0
      DO 100 I = 1, NEQNS
          CALL DSLEV1 (I,I,DIAGVL,GLOBAL,IER)
          IF ( IER .NE. 0 ) THEN
              handle error condition
          ENDIF
  100 CONTINUE
```

This matrix could also have been input strictly with DSLEVM by expanding COLSTR, ROWIND, and VALUES to include the diagonal. This action would have avoided the calls to DSLEV1. Including the diagonal entries in COLSTR and ROWIND is also acceptable to DSLEIM.

Name

DSLECO
Numeric factorization and condition number estimation

Purpose

This subprogram computes the numeric factorization and condition number estimate of the sparse symmetric matrix input to the package. This operation is not implemented for nonsymmetric matrices.

Usage

```
INTEGER*4    inrtia(3), ier
REAL*8       pvttol, rcond, global(150)
CALL DSLECO(pvttol, cond, inrtia, global, ier)
```

Input

pvttol Pivoting tolerance; $0 \leq$ **pvttol** ≤ 1. The value 0 gives a reordering that minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite.

pvttol is not used in the unsymmetric case.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

rcond Estimate of the reciprocal of the 1-norm condition number.

inrtia Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.

ier Status response:

ier = 0	Normal return.
ier = –500	Incorrect processing path; value input not performed.
ier = –501	Error in dynamic storage allocation during factorization.
ier = –502	Error in dynamic storage allocation during factorization.
ier = –503	Exactly zero pivot encountered during factorization; matrix is singular.

Example Factor the matrix input to the package and estimate its condition number. Use a pivot tolerance of 0.1.

```
INTEGER*4 INRTIA(3),IER
REAL*8    RCOND,GLOBAL(150)
CALL DSLECO (0.1D0,RCOND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEFA
Numeric factorization

Purpose

This subprogram computes the numeric factorization of the sparse symmetric matrix input to the package. No condition number estimation is performed.

Usage

```
INTEGER*4      inrtia(3), ier
REAL*8         pvttol, global(150)
CALL DSLEFA(pvttol, inrtia, global, ier)
```

Input

pvttol Pivoting tolerance; $0 \leq$ **pvttol** ≤ 1. The value 0 gives a reordering that minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite.

pvttol is not used in the unsymmetric case.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

inrtia Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.

ier Status response:

ier = 0	Normal return.
ier = –500	Incorrect processing path; value input not performed.
ier = –501	Error in dynamic storage allocation during factorization.
ier = –502	Error in dynamic storage allocation during factorization.
ier = –503	Exactly zero pivot encountered during factorization; matrix is singular.

Example Factor the matrix input to the package. Use a pivot tolerance of 0.1.

```
INTEGER*4 INRTIA(3),IER
REAL*8    GLOBAL(150)
CALL DSLEFA (0.1D0,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSLESL
Solve

Purpose This subprogram computes the solution for a set of right hand side vectors given a numeric factorization performed by DSLECO or DSLEFA.

Usage

```
INTEGER*4      nrhs, ldrhs, ierREAL*8 rhs(ldrhs, nrhs), global(150)
CALL DSLESL(nrhs, rhs, ldrhs, global, ier)
```

Input

nrhs Number of right-hand sides; **nrhs** > 0.

ldrhs The leading dimension of array **rhs** as declared in the calling program unit, with **ldrhs** ≥ **neqns**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

rhs On input, **rhs** contains the **nrhs** right-hand sides. On output, **rhs** has been overwritten with computed solutions.

Output

ier Status response:

ier = 0	Normal return.
ier = –600	Incorrect processing path; numeric factorization has not been performed.
ier = –601	Error in dynamic storage allocation.
ier = –602	**nrhs** ≤ 0.
ier = –603	**ldrhs** < **neqns**.

Example Given the following right-hand side vector, compute the solution given the numerical factorization computed by DSLECO or DSLEFA.

$$\text{rhs} = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

```
INTEGER*4 IER
REAL*8    RHS(6),GLOBAL(150)

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSLESL (1,RHS,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEDA
Deallocate working storage

Purpose

Deallocate working storage at the end of processing. If the program using the package is to continue execution after use of the package is completed, it is recommended that the user deallocate the dynamically-allocated working storage with subroutine DSLEDA.

Usage

INTEGER*4 ier
REAL*8 global(150)
CALL DSLEDA(global, ier)

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –200	No system exists for this global communications array.

Example

Deallocate working storage after use of package.

```
REAL*8 GLOBAL(150)
CALL DSLEDA (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSLEFF
Specify singularity treatment

Purpose

This subprogram provides a mechanism to deal with singular (or nearly singular) matrices by specifying a course of action to follow when zero or small diagonal elements are encountered in the factorization during subsequent calls to DSLEFA or DSLECO that specify **pvttol** = 0.

Usage

```
INTEGER*4       iabort, ier
REAL*8          fudge, tol, global(150)
CALL DSLEFF(iabort, fudge, tol, global, ier)
```

Input

iabort — Flag to specify when to terminate the factorization, as follows:

- **iabort** = 1 — Terminate the factorization if a diagonal element becomes smaller than –**tol**.
- **iabort** = 2 — Terminate the factorization if the absolute value of a diagonal element becomes smaller than **tol**, that is., when a diagonal element is near zero.
- **iabort** = 3 — Do not terminate the factorization on negative or near zero diagonal values. This is the default if DSLEFF has not been called.

fudge — User specified value of the fudge factor, **fudge** > 0. Used when **iabort** = 1 or **iabort** = 3. When a pivot that is smaller in absolute value than **tol** is encountered in DSLEFA or DSLECO, the near-zero pivot value is replaced by **fudge**. The index of the first such pivot is returned in non permuted order by the **ier** argument of DSLEFA or DSLECO.

The location of all of the fudged pivots can be determined if **fudge** ≤ **tol** by calling DSLERD to get the diagonal, and then searching it for values matching **fudge**.

tol — User-specified value of the tolerance. Those diagonal elements whose absolute values are less than **tol** during the numerical factorization are replaced by **fudge**.

Default value is 0 if DSLEFF has not been called

Updated	**global**	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	**ier**	Status response: **ier** = 0 Normal return. **ier** = –1000 Incorrect processing path; should be called before DSLEFA or DSLECO. **ier** = –1001 **tol** < 0.

Name

DSLEOC
Output control

Purpose

This subprogram may be called at any point after subprogram DSLEIN to alter either the output message level or the Fortran output unit number for message output.

Usage

```
INTEGER*4      msglvl, output
REAL*8         global(150)
CALL DSLEOC(msglvl, output, global)
```

Input

msglvl — Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output — Fortran logical unit number to which all output is written.

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Example

Increase message level from 1 to 2 while leaving the output unit the same at 6.

```
REAL*8 GLOBAL(150)
CALL DSLEOC (2,6,GLOBAL)
```

Name DSLEOP
Select reordering scheme

Purpose This subprogram selects the reordering scheme used by the solver. The default is to use the Multiple Minimum Degree (MMD) reordering.

Usage

```
CHARACTER*3    order
INTEGER*4      ier
REAL*8         global(150)
CALL DSLEOP(order, global, ier)
```

Input **order**

The reordering scheme used by the solver. The reordering scheme depends on the matrix structure input as follows.

For matrix structure input by matrix, use one of the following seven options:

order = 'MMD' or 'mmd'

Use the Multiple Minimum Degree ordering. This is the default used by xSLEIN—Refer to "DSLEIN" on page 345.

order = 'NAT' or 'nat'

Use natural ordering.

order = 'RCM' or 'rcm'

Use reverse Cuthill McKee (rcm) ordering.

order = '1WD' or '1wd'

Use one-way Dissection ordering.

order = 'GND' or 'gnd'

Use General Nested Dissection ordering.

order = 'MET' or 'met'

Use METIS 4.0.1 ordering.

order = 'CMD or 'cmd'

Use Constrained Minimum Degree ordering.

For matrix structure input by elements, columns, and finite elements, use one of the following four options:

order = 'MMD' or 'mmd'
: Use the Multiple Minimum Degree ordering.

order = 'NOT' or 'not'
: No internal reordering is performed; the original ordering is used in all phases of the solver.

order = 'MUL' or 'mul'
: Use the Multilevel Nested Dissection ordering. This ordering is very effective for certain applications, such as matrices from linear programming (LP) problem and 3-D finite element problems using brick elements.

order = 'OPT' or 'opt'
: Use the optimal ordering. This option runs both the MMD and Multilevel Nested Dissection orderings, and chooses the one which produces the smaller number of fill-ins in the numerical factorization.

Updated **global**

Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier**

Status response:

ier = 0
: Normal return.

ier = −1100
: Incorrect processing path; should be called before DSLEOR.

ier = −1101
: Invalid value for order.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the CALL statement may be improved by coding the order argument as 'MMD Reordering' or 'no reordering'.

Name DSLEPS
Print statistics

Purpose This subprogram prints available statistics about the original matrix, amount of work space in use, amount required for the next phase, and maximum amount used thus far. Also, this subprogram prints storage and arithmetic requirements, CPU time used, and computational rate for Cholesky factorization. The amount of information printed depends on the stage of execution. The number of lines of output ranges from 8 to 30, with the width of the lines being less than 80 characters.

Usage

REAL*8 global(150)
CALL DSLEPS(global)

Input

global Global communications array for this problem.

Example Print the statistics after the package has completed a numeric solution (either after DSLESL or DSLEFS).

```
REAL*8 GLOBAL(150)
CALL DSLEPS (GLOBAL)
```

Name

DSLERD
Return diagonal elements of matrix in its current form

Purpose

This subprogram is called after the matrix value input is complete. It returns the diagonal elements of the sparse matrix in its current form:

- If the matrix has not been factored, diag (A) is returned.
- If the matrix has been factored, diag (D) is returned.

The diagonal elements can be returned in the original order or in the permuted order if reordering has been completed.

Usage

```
CHARACTER*1   job
INTEGER*4     ier
REAL*8        diag(neqns), global(150)
CALL DSLERD(job, diag, global, ier)
```

Input

job Option flag:

job = 'O' or 'o'	Return the diagonal elements of the current form of the matrix in the original order.
job = 'P' or 'p'	Return the diagonal elements of the current form of the matrix in the permuted order determined by the reordering algorithm.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

diag Array of diagonal elements of the current form of the sparse matrix.

ier Status response:

ier = 0	Normal return.
ier = –800	Incorrect processing path; matrix value input not completed. Diagonal values are not returned.
ier = –801	Invalid value for job.

Notes

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the CALL statement may be improved, for example, by coding the job argument as 'Original Order' for 'O', or 'Permuted Order' for 'P'.

5 Sparse Eigenvalues and Eigenvectors

Overview

This chapter describes state-of-the-art software for solving sparse symmetric and generalized symmetric eigenvalue problems. This package of subprograms provides efficient use of the Hewlett-Packard architecture in conjunction with powerful techniques for using the sparsity of the problem to reduce the cost of solution. Accuracy is assured through appropriate numerical techniques.

This chapter explains how to use sparse eigenvalue subprograms to solve sparse eigenvalue problems where the matrix or matrices are symmetric. Subprograms are provided to:

- Solve ordinary symmetric sparse eigenvalue problems of the form $Ax=\lambda x$
- Solve certain generalized symmetric sparse eigenvalue problems of the form $Ax=\lambda Bx$

This sparse matrix eigenvalue software is designed so that it is possible to call a single subprogram to solve a sparse symmetric eigenvalue problem. However, this requires a particular format for the sparse matrix. This package provides other approaches that provide a very general interface to alternative representations of the sparse matrix. These optional approaches, however, require the user to call a sequence of subprograms.

Chapter objectives

After you read this chapter you will:

- Understand what a sparse matrix is
- Understand how to use these subprograms to compute some of the eigenvalues and eigenvectors of sparse symmetric matrices

Associated documentation

The following documents provide supplemental material for this chapter:

Grimes, R.G., J.G. Lewis and H.D. Simon. "Eigenvalue Problems and Algorithms in Structural Engineering." *Large Scale Eigenvalue Problems*. North-Holland. 1986. pp. 81-95.

Grimes, R.G., J.G. Lewis and H.D. Simon. "The Implementation of a Block Shifted and Inverted Lanczos Algorithm for Eigenvalue Problems in Structural Engineering." *Boeing Computer Services Technical Report, ETA-TR-39*. August, 1986.

Nour-Omid, B., B.N. Parlett, T. Ericsson and P.S. Jensen. "How to Implement the Spectral Transformation," *Mathematics of Computation*. 1987. 48, 178, pp. 663-673.

Parlett, B.N. *The Symmetric Eigenvalue Problem*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1980.

What you need to know to use these subprograms

Generalized symmetric eigenproblems

This package is capable of solving ordinary symmetric eigenvalue problems $Ax=\lambda x$ and certain generalized symmetric eigenvalue problems $Ax=\lambda Bx$. In the latter case, both A and B are symmetric. The ordinary symmetric eigenproblem has the property that all eigenvalues are real and that there are n real orthogonal real eigenvectors. This latter condition is equivalent to $x_i^T x_i = 1$ and $x_i^T x_j = 0$ for all eigenvectors x_i and x_j with $i \neq j$.

To have similar properties in the generalized case, more conditions are needed than just symmetry. When B is singular, there are fewer than n finite eigenvalues. If, however, there exists a positive definite matrix $C=\alpha A+\beta B$ for some scalars α and β, then all finite eigenvalues are real and the corresponding eigenvectors are real and C-orthogonal. That is, $x_i^T C x_i = 1$ and $x_i^T C x_j = 0$ for all eigenvectors x_i and x_j with $i \neq j$.

Unfortunately, there is no general and sparse algorithm for determining the necessary scalars α and β for a general pair of matrices A and B. The transformations used in this package require explicit knowledge of C, so this package is restricted to cases where the user can explicitly transform the problem to be in terms of C, or to the two standard cases where C is obvious; that is, when B or A alone is a positive definite matrix.

The standard names associated with these cases in structural engineering are:

- Vibration analysis—B is positive definite (A=K, the stiffness matrix; B=M, the mass matrix)
- Buckling analysis—A is positive definite and B is indefinite (A=K, the stiffness matrix; B=K_g, the geometric or differential stiffness matrix)

In both cases, the package allows the matrix subject to the definiteness condition to be a semi definite matrix. For example, the mass matrix in vibration analysis is required only to have non negative masses, not positive masses.

Allowing semi definiteness makes the package more generally useful, but also allows the user to specify a problem that is not a well-posed generalized symmetric eigenproblem. In most contexts, the user knows from scientific considerations that this is or is not a problem. Well-posed vibration problems with singular mass matrices are common. Normally, the package returns warnings about inconsistent inertia counts in ill-posed cases. However, supplying an indefinite matrix (for example, a mass matrix with negative masses) usually results in a fatal message describing a failure in the reorthogonalization modules or in inconsistent inertia counts.

Sparsity and storage formats

Sparse matrices are matrices in which most of the entries are zero. The goal of sparse matrix software is to take maximum advantage of these zero entries to reduce storage and arithmetic. Storage is reduced by not storing zero entries, and arithmetic is reduced by not performing operations on entries that are known to be zero.

It is easiest to see how to economize on storage. Suppose that an n-by-n matrix A has only nz nonzero entries. Then A could be specified completely by storing each of the nonzero values in an array of length nz that was accompanied by two integer arrays of length nz holding the corresponding row and column indices. Thus, $3nz$ storage suffices where n^2 storage is required for the corresponding dense matrix format.

Consider, for example, the following matrix:

11	0	13	14	0	0
0	22	23	0	25	0
31	32	33	0	35	0
41	0	0	44	45	0
0	52	53	54	55	0
0	0	0	0	0	66

This matrix could be represented in the format described above by three arrays, **irow**, **jcol**, and **mxvalu**, as shown in Figure 5-1.

Figure 5-1 Row and Column Index Sparse Matrix Representation

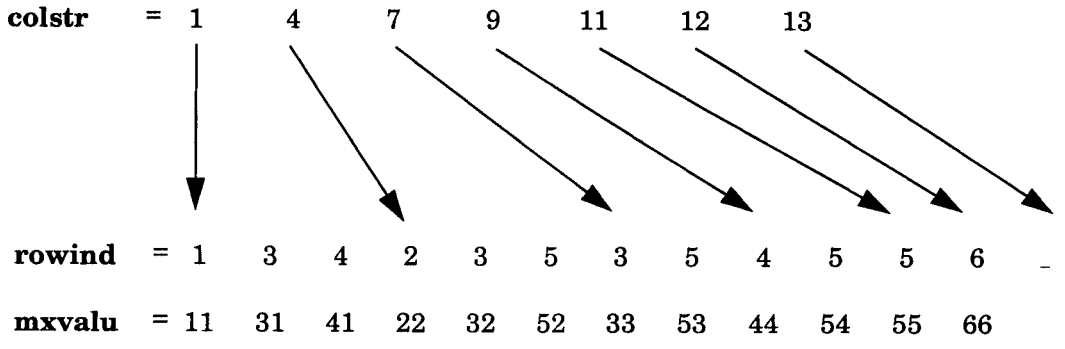
irow	=	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	4	5	6
jcol	=	1	1	1	2	2	2	3	3	3	3	4	4	4	5	5	5	5	6
mxvalu	=	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	45	55	66

In this example, the matrix entries are listed in order within each column, and the columns are listed in order, although the representation does not require that much structure. However, if the entries are required to be ordered by row and column, even less storage is needed. In addition, symmetry in the matrix can be used by storing only the entries, for example, on or below the main diagonal. Other contexts, such as finite element analysis, can make even more concise representations of the locations of the nonzeros.

This package adopts a particular internal format that allows arbitrary symmetric matrices to be stored in $2nz+n+1$ storage locations, where nz represents the number of nonzeros in the lower triangle of the matrix. In essence, the **jcol** array, which has repeated entries, is replaced by a shorter array that gives the index of the first nonzero of each column in the **mxvalu** array and an indication of the number of elements in each column. These two pieces of information per matrix column can be represented in a single array **colstr** of length $n+1$ if the convention is adopted that the number of nonzeros in column j is given by **colstr**(j+1)–**colstr**(j) and if **colstr**(n+1) is set accordingly.

This sparse matrix format, known as the *column pointer, row index* representation, is illustrated in Figure 5-2.

Figure 5-2 Column Pointer, Row Index Sparse Matrix Representation



There are three ways to communicate the coefficient matrix or matrices to the package. One is a totally general form, which allows the user to store the matrices outside the package in whatever form is most convenient. The other two ways require that the user store the matrices in a form similar to the internal format or at least with all entries in each column contiguous in memory. Any of these three can be used. However, the most general form carries additional overhead in computer time.

Description of sparse eigenvalue problems

Sparsity in the matrix does not guarantee sparsity in the matrix of eigenvectors. Indeed, it is rare that any eigenvectors of a sparse matrix are sparse. This implies that computing all the eigenvalues and eigenvectors of a sparse matrix is an enormous computation requiring storage of a large dense matrix. The common requirement for sparse eigenanalysis is to compute a subset, usually small, of the eigenvalues and vectors. This section describes how to specify the appropriate subsets to the sparse eigenanalysis package.

For notational purposes, assume that the eigenvalues λ_i are ordered $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$. The results from subroutines in this section normally form a contiguous set of eigenvalues, say, $\lambda_i, \lambda_{i+1}, \ldots, \lambda_j$. You must designate which subset you want through the following kinds of constraints, which usually relate to the application-specific meaning of the eigenvalues:

1. Eigenvalues can be restricted to lie in a finite or semi-infinite interval (find only eigenvalues in [a,b]).
2. Eigenvalues can be described by count (find only *P* eigenvalues) together with a location descriptor based either on the magnitude (either the smallest/lowest or largest/highest) or on the proximity to a fixed value (closest to *c*).

These two characterizations can be used together or separately. Examples of standard cases include:

- Find the lowest 10 eigenvalues
- Find all eigenvalues in [0.0, 100.0]
- Find the 3 eigenvalues closest to 53.1

This structure also permits combinations such as:

- Find the lowest 15 eigenvalues in [20.1, 75.3]
- Find the 10 eigenvalues closest to 15.0 in [10.0, 20.0]

The terms *lowest* and *highest* have two standard and different meanings in algebra:

- Size—for example, least magnitude
- Ordinal position—algebraically least

These subprograms follow a common engineering usage, where the interpretation is magnitude, not order. Thus, lowest means closest to zero and highest farthest from zero. The two interpretations are the same when all eigenvalues are positive, but the reverse when all eigenvalues are negative. The greatest confusion arises when the eigenvalues of interest may include both positive and negative numbers, that is, when the interval specifications include zero in the interior of the interval. To help clarify the interpretation of *lowest* in an interval containing zero in the interior, printed output from the package will describe this as *nearest* to zero.

When zero lies in the interior of the interval, the eigenvalues highest in magnitude may be drawn from both the algebraically least and the algebraically greatest. These eigenvalues could form two disjoint sets of contiguous eigenvalues rather than one. To avoid problems associated with this, the subprograms currently allow finding the highest eigenvalues only with

an interval specification [a,b] that lies to one side of zero (for example, [1.0,100.0], [–9.99,–1.0] or [0.0, 25.0], but not [–100.0,100.0]). This restriction on the interval does not apply to the other descriptors (all, lowest, or nearest).

When you want the algebraically least or greatest eigenvalues, the problem must be transformed into finding the eigenvalues nearest one or the other endpoint of the interval.

Trust regions and matrix inertias (Sturm sequence counts)

This sparse eigenanalysis package includes an independent facility for verifying that the eigenvalues computed are the eigenvalues requested. To know if a given set of 10 eigenvalues comprise the least 10 eigenvalues of a matrix, it is necessary either to compute all of the eigenvalues or to obtain information on eigenvalue counts and location through some other means.

This package uses the fact that if $A - \sigma I = LDL^T$, then the number of negative eigenvalues of D is the same as the number of eigenvalues of A that are smaller than σ. This eigenvalue package uses the sparse symmetric linear equation package to compute an LDL^T factorization, where D is a diagonal or quasi-diagonal matrix.

It is trivial to count the number of eigenvalues of each sign for D. (Technically, the three numbers giving the number of negative, zero and positive eigenvalues of D are the inertia of D and $A - \sigma I$. A related technique for tridiagonal matrices yielding so-called Sturm sequences can be extended to give the same information; this name has been carried over in certain engineering contexts to describe the inertia of the matrix.)

The matrix inertias are used in the following way: The difference between the number of negative eigenvalues for the associated D matrices for $A - \sigma_2 I$ and for $A - \sigma_1 I$ is the number of eigenvalues in the interval $[\sigma_1,\sigma_2]$. If this number agrees with the number of eigenvalues that the package has computed in this interval, all of the eigenvalues in this subinterval have been computed. We call such a subinterval a *trust region*. The goal of the package is to compute the number of eigenvalues requested by the user and to find a trust region including these eigenvalues. To this end, whenever factorizations are computed, the inertias of the associated matrix are evaluated and saved. Additional factorizations may be computed to provide trust region information, so that all returned eigenvalues are confirmed as being properly described.

This use of matrix inertia is subject to rounding errors. A particular difficulty arises when the shift value σ is actually equal to an eigenvalue. In this case $A - \sigma I$ should have at least one zero eigenvalue. In practice, the numerical value is probably very small, but nonzero. However, the matrix inertia considers only the sign, and so an incorrect count may be returned even when the factorization is stable.

The eigenvalue location counts derived from the inertias may differ from what has been computed. In such cases, a warning is issued. The algorithm attempts to avoid such situations when creating trust regions but often must use user-specified interval bounds for its final trust region. Specifying eigenvalues in an interval where the endpoints are very close to eigenvalues is likely to cause warnings.

Convention for returning eigenvalues and eigenvectors

The number of eigenvalues and vectors that will be computed by the sparse eigenanalysis package is often not known before the computation. This is obviously the case when all of the eigenvalues in an interval are requested, but it may also occur when an interval contains fewer than the number of eigenvalues requested. The eigenvector matrix is presumed to be dense, so it would occupy a large amount of memory if the number of eigenvalues computed is large. In some contexts, it may not be necessary or desirable to hold the entire eigenvector matrix in main memory. Thus, rather than request that the user supply the space for the eigenvector matrix, the eigenvectors are returned to the user through a two-pass process.

The user first calls one or more subprograms to compute the eigenvalues and eigenvectors. These inform the user of the number computed. In the second pass, the user can request that either specified single or groups of eigenvectors be returned, permitting more flexible use of storage.

This convention, requiring a second subroutine to retrieve the eigenvectors, also permits the user to have selective access to additional data that would otherwise be discarded by the algorithm. It is common for the package to compute a few more eigenvectors than the user requested. For example, the user may request the lowest 10 eigenvalues. However, if the values of the tenth, eleventh and twelfth eigenvalues are very close to one another, the package must compute all twelve eigenvalues to establish a reliable trust region containing the first 10 eigenvalues. There are other contexts, for example when the heuristic shifting strategy used is not perfect, or when some failure of the algorithm causes the computation to be cut short, where the package will compute more or different eigenvalues and vectors than requested.

The package allows the retrieval of these otherwise discarded eigenvectors if the user wants them. The main eigenanalysis subprogram returns two values:

- The number of eigenvalues and vectors computed to the specified accuracy that lie in a single reliable trust region
- The number of eigenvalues and vectors computed to the specified accuracy that do not lie in this single reliable trust region

These additional eigenvalues may or may not lie in a trust region that is an extension of the final trust region. To determine whether this is the case, the user will have to make an interpretation of the matrix inertia data, which is printed (in English) for each factorization if the output message level is set to one or greater.

Global communications array

All of the subprograms in this package use dynamic memory allocation capabilities to free users from the often difficult issue of allocating storage for the factors of the matrix. This internal storage is invisible to the user. When the sophisticated lower-level routines are used, knowledge of the internal storage is passed from subprogram to subprogram through a single communications array. This array, called **global** in each of the calling sequences, is a fixed-length double precision vector of length 150. It must not be altered by the user, as it represents the essential knowledge of the problem. Because it is the only identification for a problem, the user can handle multiple problems simultaneously by having multiple communications arrays with different names.

Error convention

Each subprogram has an error return flag, **ier**, as one of its arguments. A zero value returned in **ier** is the indication of successful processing. Fatal errors are signaled through negative values; each is a negative integer in the range $-1000 \leq \mathbf{ier} \leq -100$. The hundreds digit indicates the phase of processing in which the error occurred; the other digits specify the error itself. A positive error return indicates that some number of eigenvalues have been computed but a fatal error was encountered which caused early termination of the eigensolution phase.

In addition to the error return flag, the eigensolution phase has an additional warning return, **warnng**, to indicate when the computed results are not exactly what was requested by the user. A warning occurs, for example, when the user requests the lowest 20 eigenvalues in a finite interval which contains only 11 eigenvalues.

Output controls

This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the message level. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and thus should generally be avoided. However, this will be the proper message level when the contents of the output file must originate solely from the application program that is using the sparse matrix package. When **msglvl** = 1, a small amount of runtime statistics, including matrix inertias necessary for interpreting discardable results, and any error messages will be printed. When **msglvl** > 1, the complete set of runtime statistics will be printed. If **msglvl** ≥ 3, various arrays whose length is on the order of the number of equations will be printed. If **msglvl** ≥ 4, volumes of output will be produced for debugging purposes.

We recommend that you set **msglvl** = 1. The higher **msglvl** values (≥ 3) are intended for debugging purposes and generate copious amounts of printed output. Further, their use for this purpose may require some knowledge of the data structures being used by the package.

Paths of control

This package provides both a single standard format for sparse matrices and a completely general interface that allows the user to represent the sparse matrices in the most convenient form for use outside this package. The possible paths through the package are illustrated in Figure 5-3 and Figure 5-4.

Figure 5-3 Paths of Control—Sparse Eigenvalues and Eigenvectors

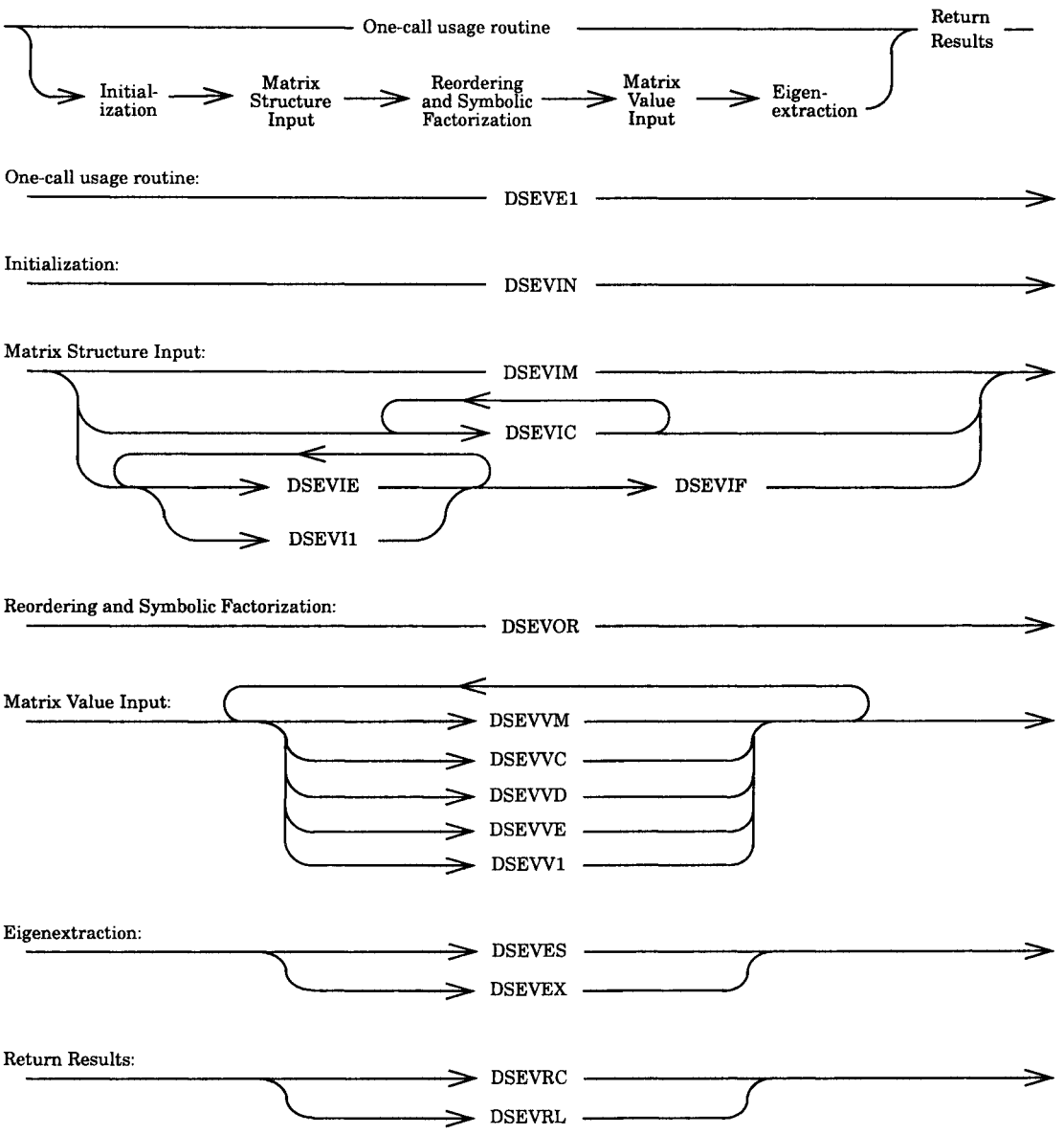
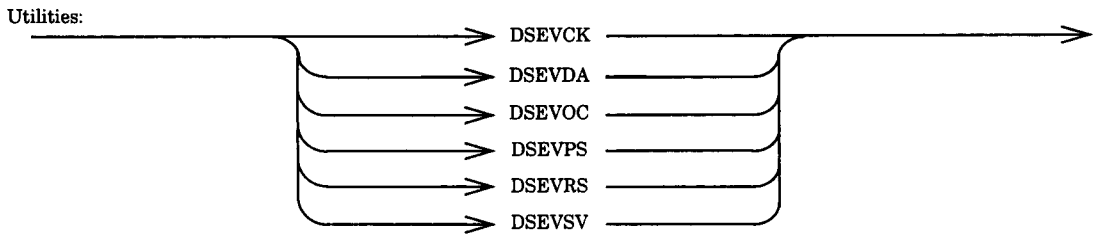


Figure 5-4 Paths of Control(continued)



To use the general interface, the user must call a sequence of subprograms that perform the following operations:

- Initialization
- Input of matrix structure
- Reordering
- Input of values of matrix entries
- Eigenextraction
- Retrieval of eigenvalues and vectors

It is possible to use this package to solve several different eigenvalue problems associated with the same matrix. This uncommon situation can be handled by calling the eigenextraction subprogram with different specifications for the desired eigenvalues. The only mechanism for solving eigenvalue problems for a sequence of matrices with different values, but the same sparsity pattern, is to use the save and restart facilities immediately after the reordering phase.

Sample program

As illustrated in Figure 5-3, there are many possible paths of control through this sparse eigenvalue package. The following sample program shows one possible path through the package, where the problem to be solved is the ordinary eigenvalue problem $Ax = \lambda x$. It is intended to demonstrate that the package is not as difficult to use as Figure 5-3 implies.

In this example, the row and column indices and the corresponding value for each nonzero entry of the matrix are stored in the three arrays IROW, JCOL, and MXVALU. This example demonstrates the use of the subroutines for solving a sparse eigenproblem when the subroutine DSEVES cannot be used. The eigenproblem posed here is to find the 10 lowest eigenvalues of the matrix A represented by the three arrays. Hence, this is an ordinary eigenproblem. The code assumes that there are NNZERO nonzero entries in the matrix, which has NEQNS rows and columns.

```
      INTEGER*4 IROW(NNZERO),JCOL(NNZERO),WARNNG
            LOGICAL*4 ORTWRN
            REAL*8    DISCRP,DUMMY,VALUE
            REAL*8    MXVALU(NNZERO),GLOBAL(150),EVALUE(10),EVECTR(NEQNS,10)

      C
      C     ... INPUT THE MATRIX STRUCTURE
      C

            CALL DSEVIN (NEQNS,'I',1,6,GLOBAL,IER )
            IF ( IER .NE. 0 ) GO TO 8000

            DO 100 K = 1, NNZERO
                I = IROW(K)
                J = JCOL(K)
                CALL DSEVI1 ('A',I,J,GLOBAL,IER)
                IF ( IER .NE. 0 ) GO TO 8000
        100 CONTINUE

            CALL DSEVIF (GLOBAL,IER)
            IF ( IER .NE. 0 ) GO TO 8000
      C
      C     ... REORDER THE MATRIX
      C

            CALL DSEVOR (GLOBAL,IER)
            IF ( IER .NE. 0 ) GO TO 8000
```

Sample program

```
C     -
C     ... INPUT MATRIX VALUES
C     -

      DO 200 K = 1, NNZERO
          I     = IROW(K)
          J     = JCOL(K)
          VALUE = MXVALU(K)
          CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  200 CONTINUE

C
C     ... COMPUTE THE EIGENVALUES AND EIGENVECTORS
C

      CALL DSEVES (10,'L','O',.FALSE.,DUMMY,.FALSE.,DUMMY,
     1             DUMMY,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
      IF ( IER .NE. 0 ) GO TO 8000

C
C     ... CHECK ACCURACY
C

      CALL DSEVCK (.TRUE.,ORTWRN,DISCRP,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
      IF ( ORTWRN ) PRINT *,'EIGENVALUE/EIGENVECTOR DISCREPANCY
=',DISCRP
C     -
C     ... RETRIEVE EIGENVALUES AND EIGENVECTORS
C     -
      CALL DSEVRC (20,EVALUE,1,NFOUND,EVECTR,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C     ... USE THE EIGENVALUES AND EIGENVECTORS
C
      .
      .
      .
C
C     ... ERROR TRAP
C

 8000  ..........
```

Subprograms for sparse eigenvalues and eigenvectors

The following sections describe subprograms included with VECLIB for solving sparse symmetric and generalized symmetric eigenvalue problems.

Name

DSEVE1
One-call usage

Purpose

This subprogram provides a one-call usage for the sparse eigenvalue package. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package described in this chapter.

Usage

CHARACTER*1	**bmxtyp, which, pbtype**
INTEGER*4	**annzer, bnnzer, norder, msglvl, output, acolst(norder+1), arowin(annzer), bcolst(norder+1), browin(bnnzer), neigvl, nfound, ndiscd, ier, warnng**
LOGICAL*4	**lfinit, rfinit**
REAL*8	**avalue(annzer), bvalue(bnnzer), lftend, rhtend, center, global(150)**

CALL DSEVE1(norder, msglvl, output, acolst, arowin, avalue, bmxtyp, bcolst, browin, bvalue, neigvl, which, pbtype, lfinit, lftend, rfinit, rhtend, center, nfound, ndiscd, global, ier, warnng)

Input

norder Order of matrices (number of degrees of freedom), **norder** > 0.

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages, summary statistics and inertia information.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output Fortran logical unit number for file to which printable output will be written.

acolst **acolst**(j) gives the address in **arowin** of the first nonzero in the lower triangular part of column j of the matrix A. All of the nonzeros for column j are found in ascending order in **arowin**(**acolst**(j)), **arowin**(**acolst**(j)+1), ..., **arowin**(**acolst**(j+1)–1).

acolst(**norder**+1) must be set to one greater than the total number of nonzeros, **annzer**, in the lower triangular part of the matrix.

arowin List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix A.

avalue List of values corresponding in position to the indices in **arowin**.

bmxtyp A character string specifying the type of sparsity found in the right-hand-side B matrix in a generalized eigenvalue problem $Ax=\lambda Bx$. Only the first character is significant, but longer input can be used for clarity (for example, 'Identity' or 'Diagonal').

'I' or 'i' or '1'	B is an identity matrix. Use this option for a standard (ordinary) eigenproblem $Ax=\lambda x$, where there is no second matrix.
'A' or 'a' or 'K' or 'k'	The sparsity structure of B is the same as that of A or K.
'D' or 'd'	B is a diagonal matrix but not necessarily the identity.
Any other letter	B is a general sparse matrix whose structure is specified independently from A.

bcolst If B is specified as a general sparse matrix with structure different than A, then **bcolst**(j) gives the ordinal in **browin** of the first nonzero in the lower triangular part of column j of the matrix B. All of the nonzeros for column j are found in ascending order in **browin**(**bcolst**(j)), **browin**(**bcolst**(j)+1), ..., **browin**(**bcolst**(j+1)–1).

bcolst(**norder**+1) must be set to one greater than the total number of nonzeros, **bnnzer** in the lower triangular part of the matrix.

Not referenced unless the matrix B is specified as anything other than a general sparse matrix with structure different from A.

browin List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix B.

Not referenced unless the matrix B is specified as anything other than a general sparse matrix with structure different from A.

bvalue If B is a general sparse matrix, then **bvalue** contains the list of values corresponding in position to the indices in **arowin** or **browin** depending on whether B has the same structure as A or not. If B is diagonal, then **bvalue** contains the diagonal entries. Not referenced if B is the identity.

neigvl Number of eigenvalues to be found, **neigvl** > 0.

which Character string indicating which eigenvalues are to be computed. Only the first character is significant, but longer input can be used for clarity (for example, 'Lowest' or 'All').

'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
'H' or 'h'	The highest (greatest magnitude) eigenvalues.
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest **center**.
'A' or 'a'	All eigenvalues in the specified interval.

pbtype Character string indicating type of problem. Only the first character is significant, but longer input can be used for clarity (for example, 'Vibration' or 'Ordinary').

'V' or 'v'	Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite.
'B' or 'b'	Generalized symmetric (buckling) problem $Kx = \lambda K_\delta x$, with K positive semi definite and K_δ possibly indefinite.
'O' or 'o'	Ordinary symmetric eigenproblem $Kx = \lambda x$.

lfinit If .TRUE., the value of the argument **lftend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.

lftend Left endpoint of an interval in which the desired eigenvalues lie; not used if **lfinit** is .FALSE.

rfinit If .TRUE., the value of the argument **rhtend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.

rhtend Right endpoint of an interval in which the desired eigenvalues lie; not used if **rfinit** is .FALSE.

center A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when **which** specifies 'centered' or 'nearest'.

Output

nfound The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.

ndiscd The number of other eigenvalues computed during the course of the computation, but discarded because they were outside of the specifications for eigenvalues or because they were not confirmed by inertia computations.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier Status response:

ier = 0	Normal return.
ier < 0	Fatal error, no useful results returned.
ier > 0	Fatal error, but some eigenvalues and vectors computed.
ier = –101	Error in dynamic storage allocation during matrix structure input.
ier = –102	**norder** ≤ 0.
ier = –201	Error in dynamic storage allocation during ordering.
ier = –301	Error in dynamic storage allocation.
ier = –302	Internal error.
ier = –401	Error in dynamic storage allocation.
ier = ±701	Error in storage allocation.
ier = ±702	Illegal specification for **lftend** and **rhtend**.
ier = ±703	Illegal specification for **pbtype**.
ier = ±704	Illegal specification for **which**.
ier = ±705	**which** = **highest** for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.

	ier = ±724	QL iteration did not converge.
	ier = ±725	Number of eigenvalues computed exceeded storage limitations.
	ier = ±726	Internal error involving access of Lanczos recurrence vectors.
	ier = ±727	Singular Value Decomposition did not converge.
	ier = ±728	Gram-Schmidt reorthogonalization did not converge.
	ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
	ier = ±730	Number of trust regions formed exceeded storage limitations.
	ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
	ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.
warnng	Warning status. **warnng** includes several different warnings encoded in a three-digit decimal integer:	
	low-order digit:	
	warnng = xx0	Normal return.
	warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
	warnng = xx2	Fewer modes computed than requested because eigenvalues are wildly different in size.
	warnng = xx3	Both of the above conditions occurred.
	ten's digit:	
	warnng = x0x	Normal return.
	warnng = x1x	At some point in the process, an inconsistent inertia count was found.
	hundred's digit:	
	warnng = 0xx	Normal return.

warnng = 1xx Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example In a small (order 6) sparse generalized eigenvalue problem, the matrices A and B are as follows:

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix}$$

$$B = \begin{matrix} 1.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix}$$

The eigenproblem is to compute the two eigenvalues nearest zero.

```
INTEGER*4 MSGLVL,OUTPUT,ACOLST(7),AROWIN(13),IER
REAL*8    GLOBAL(150),AVALUE(13),BVALUE(13)

MSGLVL = 1
OUTPUT = 6

ACOLST(1) = 1
ACOLST(2) = 4
ACOLST(3) = 7
ACOLST(4) = 9
ACOLST(5) = 12
ACOLST(6) = 13
ACOLST(7) = 14

AROWIN(1) = 1
AROWIN(2) = 3
AROWIN(3) = 4
AROWIN(4) = 2
AROWIN(5) = 3
AROWIN(6) = 5
```

```
AROWIN(7) = 3
AROWIN(8) = 5
AROWIN(9) = 4
AROWIN(10) = 5
AROWIN(11) = 6
AROWIN(12) = 5
AROWIN(13) = 6

AVALUE(1) = 4.0
AVALUE(2) = 1.0
AVALUE(3) = 1.0
AVALUE(4) = 4.0
AVALUE(5) = 1.0
AVALUE(6) = 1.0
AVALUE(7) = 4.0
AVALUE(8) = 1.0
AVALUE(9) = 4.0
AVALUE(10) = 1.0
AVALUE(11) = 1.0
AVALUE(12) = 4.0
AVALUE(13) = 1.0

BVALUE(1) = 1.0
BVALUE(2) = 1.0
BVALUE(3) = 1.0
BVALUE(4) = 1.0
BVALUE(5) = 1.0
BVALUE(6) = 1.0
BVALUE(7) = 1.0
BVALUE(8) = 1.0
BVALUE(9) = 1.0
BVALUE(10) = 1.0
BVALUE(11) = 1.0
BVALUE(12) = 1.0
BVALUE(13) = 1.0

CALL DSEVE1 (6,MSGLVL,OUTPUT,ACOLST,AROWIN,AVALUE,
         'A',IDUMMY,IDUMMY,BVALUE,2,'LOWEST',
         'VIBRATION',.FALSE.,-1.0D30,.FALSE.,1.0D30,0.0D0,
         NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVIN
Initialize sparse eigenvalues/eigenvectors

Purpose

This subprogram provides information to start the sparse eigenvalue package.

Usage

```
CHARACTER*1   bmxtyp
INTEGER*4     norder, msglvl, output, ier
REAL*8        global(150)
CALL DSEVIN(norder, bmxtyp, msglvl, output, global, ier)
```

Input

norder Order of matrices (number of degrees of freedom), **norder** > 0.

bmxtyp A character string specifying the type of sparsity found in the right-hand-side B matrix in a generalized eigenvalue problem $Ax = \lambda Bx$.

'I' or 'i' or '1'	B is an identity matrix. Use this option for a standard (ordinary) eigenproblem $Ax = \lambda x$, where there is no second matrix.
'A' or 'a' or 'K' or 'k'	The sparsity structure of B is the same as that of A or K.
'D' or 'd'	B is a diagonal matrix, but not necessarily the identity.
Any other letter	B is a general sparse matrix whose structure is specified independently from A.

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages, summary statistics and inertia information.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output Fortran logical unit number to which all printable output will be written.

Output

global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.	
ier	Status response:	
	ier = 0	Normal return.
	ier = –101	Error in dynamic storage allocation.
	ier = –102	**norder** not positive.

Notes

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **bmxtyp** argument as 'identity' or 'diagonal'.

Example

Prepare to compute some of the eigenvalues and eigenvectors of a 10,000 by 10,000 matrix (ordinary symmetric for eigenproblem). Obtain error messages and standard minimal output on Fortran logical unit 6.

```
INTEGER*4 NORDER, IER
REAL*8    GLOBAL(150)
NORDER = 10000
CALL DSEVIN (NORDER,'IDENTITY',1,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVI1
Matrix structure input by single entry

Purpose

This subprogram adds a single entry in the (**irow, jcol**) position in the lower triangle to the set of known nonzeros for one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     irow, jcol, ier
REAL*8        global(150)
CALL DSEVI1(matrix, irow, jcol, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **norder**, where **norder** is the matrix order specified in the call to DSEVIN. Input of diagonal entries is optional.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **norder**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –104	Illegal value for **jcol**.
ier = –105	Illegal value for **irow**.

ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to *B*, which was not specified as having a general sparse structure (*B* is a diagonal matrix, an identity matrix, or has same structure as *A*).

Notes Calls to DSEVI1 and DSEVIE can be intermixed. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix *A*.

```
INTEGER*4 I,J,IER
REAL*8    GLOBAL(150)
I = 3035
J = 1024
CALL DSEVI1 ('A',I,J,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVIC
Matrix structure input by column

Purpose

This subprogram adds a list of indices in a single column in the lower triangle to the set of known nonzeros for one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     jcol, nzcol, jrowin(nzcol), ier
REAL*8        global(150)
CALL DSEVIC(matrix, jcol, nzcol, jrowin, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

jcol Column index, $1 \le$ **jcol** $\le$ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN. Columns must be presented in order from 1 to **norder**, except that columns with no entries can be skipped.

nzcol Number of nonzeros in column **jcol** of the matrix, **nzcol** ≥ 0.

jrowin List of row indices for all nonzeros, in ascending order, in the lower triangle part of column **jcol** of the matrix designated by **matrix**, **jcol** $\le$ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) $\le$ **norder**.

Input of diagonal entries is optional.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –103	Illegal value for **nzcol**.
ier = –104	Illegal value for **jcol**; either out of range or out of order.
ier = –106	Illegal value for at least one entry in **jrowin** or entries out of order.
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to *B*, which was not specified as having a general sparse structure (*B* is a diagonal matrix, an identity matrix, or has same structure as *A*).

Notes

The entire matrix structure must be input with DSEVIC if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIM.

If the matrix entries are available by column, using DSEVIC is more efficient than using DSEVI1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix *A*. Columns 1 to 4518 already have been input to the package using DSEVIC.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)
J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004
CALL DSEVIC ('A',J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVIE
Matrix structure input by finite element

Purpose

This subprogram adds indices to the set of known nonzeros in the lower triangle of one of the sparse matrices corresponding to a finite element or clique.

Usage

```
CHARACTER*1   matrix
INTEGER*4     nnode, nodlst(nnode), ier
REAL*8        global(150)
CALL DSEVIE(matrix, nnode, nodlst, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

nnode Number of nodes in the finite element or clique, $1 \le$ **nnode** $\le$ **norder**, where **norder** is the matrix order specified in the call to DSEVIN.

nodlst List of nodes in element or clique, $1 \le$ **nodlst**(i) $\le$ **norder** for all i, and **nodlst**(i) $\ne$ **nodlst**(j) for all i and j with $i \ne j$. All pairs (**nodlst**(i),**nodlst**(j)) in the lower triangle are added to the sparsity structure of the matrix designated by **matrix**.

Updated

nodlst The order of the values in **nodlst** may be changed.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –107	Illegal value for **nnode**.
ier = –108	Illegal value for at least one entry in **nodlst**.
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to *B*, which was not specified as having a general sparse structure (*B* is a diagonal matrix, an identity matrix, or has same structure as *A*).

Notes Calls to DSEVIE can be intermixed with calls to DSEVI1. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of *A*. Add positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix *A*.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989
CALL DSEVIE ('A',NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVIM
Matrix structure input by matrix

Purpose

This subprogram specifies the locations of all of the nonzeros in the lower triangle of one of the matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     norder, nnzero, colstr(norder+1), rowind(nnzero), ier
REAL*8        global(150)
CALL DSEVIM(matrix, colstr, rowind, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

colstr **colstr**(*j*) gives the address in **rowind** of the first nonzero in the lower triangular part of column *j* of the matrix specified by **matrix**. All of the nonzeros for column *j* are found in ascending order in **rowind**(**colstr**(*j*)), **rowind**(**colstr**(*j*)+1), ..., **rowind**(**colstr**(*j*+1)–1).

colstr(**norder**+1) must be set to one greater than the total number of nonzeros, **nnzero** in the lower triangular part of the matrix, where **norder** is the matrix order as specified in the call to DSEVIN.

rowind List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix designated by **matrix**. Input of diagonal entries is optional.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –106	Illegal value for at least one entry in **colstr** (not increasing or negative) or invalid entries in **rowind** (entries out of order within a column, or out of the lower triangle).
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to *B*, which was not specified as having a general sparse structure (*B* is a diagonal matrix, an identity matrix, or has same structure as *A*).

Notes This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure for both matrices must be input with DSEVIM if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIC.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, B is a mass matrix with the following structure:

$$\begin{matrix} x & 0 & x & x & 0 & 0 \\ 0 & x & x & 0 & x & 0 \\ x & x & x & 0 & x & 0 \\ x & 0 & 0 & x & x & 0 \\ 0 & x & x & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

```
INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    GLOBAL(150)
COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

CALL DSEVIM ('B',COLSTR,ROWIND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVIF
End of matrix structure input

Purpose This subprogram indicates the end of structure input for matrices in the sparse eigenvalue problem. DSEVIF is used only if subprograms DSEVI1 and/or DSEVIE were the mechanism by which the structure was input.

Usage

```
INTEGER*4       ier
REAL*8          global(150)
CALL DSEVIF(global, ier)
```

Updated

global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier	Status response:	
	ier = 0	Normal return.
	ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
	ier = –101	Error in dynamic storage allocation.

Example The nonzero structure of a pair of finite element matrices was passed to the sparse eigenvalue package using repeated calls to subroutine DSEVIE. Signal that no more nonzeros will be added to either matrix.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSEVIF (GLOBAL,IER )
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVOR
Reordering and symbolic factorization

Purpose

This subprogram reorders the matrix whose pattern of nonzeros is given by the locations where either *A* or *B* is nonzero, such that an efficient sparse factorization of such a matrix can be obtained. DSEVOR then constructs data structures required for the sparse factorization.

Usage

```
INTEGER*4      ier
REAL*8         global(150)
CALL DSEVOR(global, ier)
```

Output

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier — Status response:

ier = 0	Normal return.
ier = –200	Incorrect processing path; structure input not completed, or value input subroutines already called.
ier = –201	Error in dynamic storage allocation.
ier = –301	Error in dynamic storage allocation.
ier = –302	Illegal error.

Example

The structure of the sparse matrices in a sparse eigenvalue problem have been communicated to the eigenanalysis package using DSEVIN followed by DSEVIM or DSEVIC or the following: DSEVI1, DSEVIE and DSEVIF. The next step is the process to obtain good reordering for the factorizations that will be needed in the eigenextraction process.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSEVOR (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVV1
Matrix value input by single entry

Purpose This subprogram adds to the value of the entry in the (**irow, jcol**) position in the lower triangle of one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     irow, jcol, ier
REAL*8        value, global(150)
CALL DSEVV1(matrix, irow, jcol, value, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **norder**.

value Numeric value that will be added to any previous values input for this location.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –402	Subscript pair (**irow, jcol**) is not in lower triangle of matrix.
ier = –403	Illegal value for matrix designator.
ier = –404	Subscript pair (irow, jcol) was not specified in structure input. No room for value.

ier = –405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix A.

```
INTEGER*4 I,J,IER
REAL*8    VALUE,GLOBAL(150)
I = 3035
J = 1024
VALUE = 4.523D-5
CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVVC
Matrix value input by column

Purpose

This subprogram adds to the values of a list of nonzero entries in the lower triangle of a single column of one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     jcol, nzcol, jrowin(nzcol), ier
REAL*8        values(nzcol), global(150)
CALL DSEVVC(matrix, jcol, nzcol, jrowin, values, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

jcol Column index, $1 \leq$ **jcol** $\leq$ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

nzcol Number of values in the list to be added to column **jcol** of the matrix, **nzcol** > 0.

jrowin List of row indices in ascending order for values to be added to the lower triangle part of column **jcol** of the matrix designated by **matrix**, **jcol** $\leq$**jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) $\leq$ **norder**. It is not necessary that all entries in column **jcol** be included in **jrowin**.

values List of values corresponding to positions specified by **jcol** and **jrowin**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.

ier = –402	Illegal value for either **nzcol** or **jcol**.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair (**jrowin(j),jcol**) was not specified in structure input. No room for value.
ier = –405	The *B* matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The *B* matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries are available by column, using DSEVVC is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix *A*.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 6

JROWIN(1) = 4519
JROWIN(2) = 4520
JROWIN(3) = 4521
JROWIN(4) = 6000
JROWIN(5) = 6002
JROWIN(6) = 6004

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0

CALL DSEVVC ('A',J,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVVD
Matrix value input to main diagonal

Purpose This subprogram adds to the values of corresponding entries on the main diagonal of one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     norder, ier
REAL*8        values(norder), global(150)
CALL DSEVVD(matrix, values, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

values Array of values to be added to the corresponding entries on main diagonal of the matrix specified by **matrix**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair **(jrowin(j),jcol)** was not specified in structure input. No room for value.
ier = –406	The *B* matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries of the main diagonal are available as a vector, using DSEVVD is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Create a main diagonal equal to 4.0 for *B*.

```
INTEGER*4 IER
REAL*8    VALUES(100),GLOBAL(150)

VALUES(1) = 4.0D0
VALUES(2) = 4.0D0
VALUES(3) = 4.0D0
VALUES(4) = 4.0D0
VALUES(5) = 4.0D0
VALUES(6) = 4.0D0

CALL DSEVVD ('B',VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVVE
Matrix value input by finite element

Purpose

This subprogram adds to the values of a set of nonzero entries in the lower triangle of one of the sparse matrices corresponding to a finite element or clique.

Usage

```
CHARACTER*1   matrix
INTEGER*4     nnode, ldelmx, nodlst(nnode), ier
REAL*8        elmmtx(ldelmx, nnode), global(150)
CALL DSEVVE(matrix, nnode, nodlst, elmmtx, ldelmx, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

nnode Number of nodes in the finite element or clique, $1 \le$ **nnode** $\le$ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

nodlst List of nodes in element or clique, $1 \le$ **nodlst**(i) $\le$ **norder** for all i, and **nodlst**(i) $\ne$ **nodlst**(j) for all i and j with $i \ne j$. Values for all pairs (**nodlst**(i),**nodlst**(j)) in the lower triangle are added to the values of the matrix designated by **matrix**.

elmmtx A two-dimensional array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of **elmmtx** is referenced. The value in **elmmtx**(k,l) is added to the value in position (**nodlst**(k),**nodlst**(l)) in the sparse matrix.

ldelmx The leading dimension of array **elmmtx** as declared in the calling program unit, with **ldelmx** $\ge$ **nnode**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.

ier = –401	Error in dynamic storage allocation.
ier = –402	Illegal value for either **nnode** or in **nodlst**.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair (**nodlst(k),nodlst(l)**) was not specified in structure input. No room for value.
ier = –405	The *B* matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The *B* matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed. If the matrix entries are available as a sum of elemental matrices, using DSEVVE is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Rows and columns 345, 346, 347, and 989 form a small dense submatrix of *A*. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in matrix *A*.

```
      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8    ELMMTX(10,10),GLOBAL(150)

      NNODE = 4

      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989

      DO 200 K = 1, NNODE
          DO 100 L = K, NNODE
              ELMMTX(K,L) = 1.0D0
  100     CONTINUE
  200 CONTINUE
      CALL DSEVVE ('A',NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
          handle error condition
      ENDIF
```

Name DSEVVM
Matrix value input by matrix

Purpose This subprogram adds values to all of the entries in the lower triangle of one of the sparse matrices.

Usage

```
CHARACTER*1   matrix
INTEGER*4     norder, nnzero, colstr(norder+1), rowind(nnzero), ier
REAL*8        values(nnzero), global(150)
CALL DSEVVM(matrix, colstr, rowind, values, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

colstr **colstr**(*j*) gives the address in **rowind** of the first nonzero in the lower triangular part of column *j* of the matrix specified by **matrix**. All of the nonzeros for column *j* are found, in ascending order, in **rowind**(**colstr**(*j*)), **rowind**(**colstr**(*j*)+1), ..., **rowind**(**colstr**(*j*+1)–1).

colstr(**norder**+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the lower triangular part of the matrix, where **norder** is the matrix order as specified in the call to DSEVIN.

rowind List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix designated by **matrix**. Diagonal entries may be present, but are not required.

values List of values corresponding in position to the indices in **rowind**. These values will be added to any values already present in the matrix specified by **matrix**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –402	Illegal value for either **nnode** or in **nodlst**.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair was not specified in structure input. No room for value.
ier = –405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes This is the most efficient mechanism for specifying nonzero values. Normally, DSEVVM is used in conjunction with DSEVIM. The matrix structure passed to DSEVVM must be identical to the structure passed to DSEVIM.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, B is the following mass matrix:

4.0	0.0	1.0	1.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	0.0	1.0	0.0
1.0	0.0	0.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Use DSEVVM and DSEVVD to input this as B.

```
INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    VALUES(6),DIAGNL(6),GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 6
ROWIND(6) = 6

VALUES(1) = 1.0
VALUES(2) = 1.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 1.0

CALL DSEVVM ('B',COLSTR,ROWIND,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

DIAGNL(1) = 4.0
DIAGNL(2) = 4.0
DIAGNL(3) = 4.0
DIAGNL(4) = 4.0
DIAGNL(5) = 4.0
DIAGNL(6) = 0.0
CALL DSEVVD ('B',DIAGNL,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

NOTE It would not have been necessary to use DSEVVD if the diagonal values had been held in the same value array as the rest of the sparse matrix.

Name

DSEVES
Eigenextraction

Purpose

This subprogram computes selected eigenvalues and eigenvectors of a sparse symmetric matrix A or of a sparse symmetric matrix pencil (A,B). The matrix or matrices have already been processed through the structure input, reordering, and value input phases. This is the standard interface; subroutine DSEVEX provides additional control parameters.

Usage

```
CHARACTER*1   which, pbtype
INTEGER*4     neigvl, nfound, ndiscd, ier, warnng
LOGICAL*4     lfinit, rfinit
REAL*8        lftend, rhtend, center, global(150)
CALL DSEVES(neigvl, which, pbtype, lfinit, lftend, rfinit, rhtend, center,
nfound, ndiscd, global, ier, warnng)
```

Input

neigvl Number of eigenvalues to be found, **neigvl** > 0.

which Character string indicating which eigenvalues are to be computed.

'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
'H' or 'h'	The highest (greatest magnitude) eigenvalues.
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest **center**.
'A' or 'a'	All eigenvalues in the specified interval.

pbtype Character string indicating type of problem.

'V' or 'v'	Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite.
'B' or 'b'	Generalized symmetric (buckling) problem $Kx = \lambda K_{\delta}x$, with K positive semi definite and K_{δ} possibly indefinite.
'O' or 'o'	Ordinary symmetric eigenproblem $Kx = \lambda x$.

	lfinit	If .TRUE., the value of the argument **lftend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.
	lftend	Left endpoint of an interval in which the desired eigenvalues lie; not used if **lfinit** is .FALSE.
	rfinit	If .TRUE., the value of the argument **rhtend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.
	rhtend	Right endpoint of an interval in which the desired eigenvalues lie; not used if **rfinit** is .FALSE.
	center	A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when **which** specifies 'centered' or 'nearest'.
Updated	**global**	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	**nfound**	The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.
	ndiscd	The number of other eigenvalues computed during the course of the computation but discarded because they were outside specifications for eigenvalues or because they were not confirmed by inertia computations.
	ier	Status response:

ier = 0	Normal return.
ier < 0	Fatal error, no useful results returned.
ier > 0	Fatal error, but some eigenvalues and vectors computed.
ier = –700	DSEVEX called without first successfully calling earlier stages.
ier = ±701	Error in storage allocation.
ier = ±702	Illegal specification for **lftend** and **rhtend**.

ier = ±703	Illegal specification for **pbtype**.
ier = ±704	Illegal specification for **which**.
ier = ±705	**which** = **highest** for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.
ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ±730	Number of trust regions formed exceeded storage limitations.
ier = ±731	Error during restoration of a numeric factorization.
ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.

	ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.
warnng	Warning status. **warnng** includes several different warnings encoded in a three-digit decimal integer:	
	low-order digit:	
	warnng = xx0	Normal return.
	warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
	warnng = xx2	Fewer modes computed than requested because eigenvalues are different in size.
	warnng = xx3	Both of the above conditions occurred.
	ten's digit:	
	warnng = x0x	Normal return.
	warnng = x1x	At some point in the process, an inconsistent inertia count was found.
	hundred's digit:	
	warnng = 0xx	Normal return.
	warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example 1 Compute the lowest 10 eigenvalues of a structural engineering vibration analysis, where matrix A is the stiffness matrix K, matrix B is the mass matrix M, and matrices K and M are positive definite or semi definite.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (10,'LOWEST','VIBRATION',.FALSE.,-1.0D30,
         .FALSE,.1.0D30,1.0D30,NFOUND,NDISCD,GLOBAL,IER,
         WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Example 2 Compute the lowest negative eigenvalue and corresponding eigenvector for a structural engineering buckling analysis, where the matrix A is the stiffness matrix K and the matrix B is the geometric stiffness matrix K_δ. The matrix K is positive semi definite.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (1,'LOWEST','BUCKLING',.FALSE.,-1.0D30,
         .TRUE.,0.0D0,0.0D0,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Example 3 Compute the 200 eigenvalues and corresponding eigenvectors closest to 3.14159 for a sparse matrix A.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (1,'NEAREST','ORDINARY',.FALSE.,-1.0D30,
         .FALSE.,1.0D30,3.14159D0,NFOUND,NDISCD,GLOBAL,IER,
         WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Name

DSEVEX
Eigenextraction

Purpose

This subprogram computes selected eigenvalues and eigenvectors of a sparse symmetric matrix A or of a sparse symmetric matrix pencil (A,B). The matrix or matrices have already been processed through the structure input, reordering, and value input phases. This is the sophisticated interface; subroutine DSEVES is simpler to use.

DSEVEX allows the user to control three parameters, the block size, the accuracy tolerance, and the shifting scale, that are selected heuristically by DSEVE1. DSEVEX also provides a mechanism for giving initial guesses for the eigenvectors. Each of these parameters is described in more detail in "Usage". Users with a great deal of experience in particular applications may find more efficient settings for these parameters than the general defaults in the code. However, it should be noted that the performance of the package can be degraded significantly by poor choices.

Usage

CHARACTER*1	**which, pbtype**
INTEGER*4	**neigvl, mxbksz, nusrvc, nfound, ndiscd, ier, warnng**
LOGICAL*4	**lfinit, rfinit**
REAL*8	**lftend, rhtend, center, tolact, shfscl, global(150), usrsvc(norder, nusrvc)**

CALL DSEVEX(neigvl, which, pbtype, lfinit, lftend, rfinit, rhtend, center, mxbksz, tolact, shfscl, nusrvc, usrsvc, nfound, ndiscd, global, ier, warnng)

Input

neigvl Number of eigenvalues to be found.

which Character string indicating which eigenvalues are to be computed:

'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
'H' or 'h'	The highest (greatest magnitude) eigenvalues.
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest **center**.
'A' or 'a'	All eigenvalues in the specified interval.

pbtype Character string indicating type of problem:

'V' or 'v'	Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi definite.

	'B' or 'b'	Generalized symmetric (buckling) problem $Kx = \lambda K_\delta x$, with K positive semi definite and K_δ possibly indefinite.
	'O' or 'o'	Ordinary symmetric eigenproblem $Kx = \lambda x$.
lfinit		If .TRUE., the value of the argument **lftend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.
lftend		Left endpoint of an interval in which the desired eigenvalues lie; not used if **lfinit** is .FALSE.
rfinit		If .TRUE., the value of the argument **rhtend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.
rhtend		Right endpoint of an interval in which the desired eigenvalues lie; not used if **rfinit** is .FALSE.
center		A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when **which** specifies 'centered' or 'nearest'.
mxbksz		A limit on the block size used in the block Lanczos recurrence. In general, it is best to choose a block size as large as, or slightly larger than, the largest multiplicity of the eigenvalues desired. The following constraints also apply: • A block size of 1 is usually less efficient than a block size of 2. • Block sizes larger than 10 are usually less efficient than smaller block sizes. • If **mxbksz** is zero or negative, a default block size optimized to the computer system will be chosen. In all cases, the Lanczos algorithm may find it necessary to reduce the block size to fit the problem into the storage available.

	tolact	An accuracy tolerance for the eigenvalues, which must be in the range from $\sqrt{\epsilon} \leq$ **tolact** $\leq \epsilon$, where ϵ is the machine precision. If a negative or zero value is supplied, a default value of $\epsilon^{2/3}$ will be used. Special care should be taken if the value of **tolact** is reduced below its default value. In particular, it is important to be sure that the computed eigenvectors remain adequately orthogonal. Subroutine DSEVCK can facilitate such monitoring.
	shfscl	An estimate of the magnitude of the smallest nonzero eigenvalues. The package will compute a heuristic estimate if the user does not supply an estimate (signaled by **shfscl** ≤ 0.0). This argument is provided only to assist the algorithm in cases where zero lies in the interval of interest or when it is not appropriate to use a shift of 0.0 either because A or K is singular (has rigid body modes) or because this is a buckling analysis.
	nusrvc	Number of user starting vectors provided. Any approximate eigenvectors supplied by the user will be used to create a starting block for the Lanczos algorithm. Provision of starting vectors can be effective when only a very few eigenvalues are to be computed, but it has limited effect otherwise.
	usrsvc	Array containing the starting vectors, one per column. Referenced only if **nusrvc** is positive. **norder** is the problem size as set in DSEVIN.
Updated	**global**	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	**nfound**	The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.
	ndiscd	The number of other eigenvalues computed during the course of the computation but discarded because they were outside of the specifications for eigenvalues or because they were not confirmed by inertia computations.

ier Status response:

ier = 0	Normal return.
ier < 0	Fatal error, no useful results returned.
ier > 0	Fatal error, but some eigenvalues and vectors computed.
ier = –700	DSEVEX called without first successfully calling earlier stages.
ier = ±701	Error in storage allocation.
ier = ±702	Illegal specification for **lftend** and **rhtend**.
ier = ±703	Illegal specification for **pbtype**.
ier = ±704	Illegal specification for **which**.
ier = ±705	**which** = **highest** for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±709	Starting block I/O error.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.

	ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
	ier = ±730	Number of trust regions formed exceeded storage limitations.
	ier = ±731	Error during restoration of a numeric factorization.
	ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
	ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.
	ier = ±742	Error in deallocation of dynamic storage.
warnng	Warning status. **warnng** includes several different warnings encoded in a three-digit decimal integer:	
	low-order digit:	
	warnng = xx0	Normal return.
	warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
	warnng = xx2	Fewer modes computed than requested because eigenvalues are different in size.
	warnng = xx3	Both of the above conditions occurred.
	ten's digit:	
	warnng = x0x	Normal return.
	warnng = x1x	At some point in the process, an inconsistent inertia count was found.
	hundred's digit:	
	warnng = 0xx	Normal return.
	warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example 1 Compute the lowest 10 eigenvalues of a structural engineering vibration analysis, where the matrix A is the stiffness matrix K, the matrix $B$is the mass matrix M, and both matrices K and M are positive definite or semi-definite.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVEX (10,'LOWEST','VIBRATION',.FALSE.,-1.0D30,
         .FALSE,.1.0D30,1.0D30,0,0.0D0,0.0D0,0,0.0D0,
         NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Example 2 Compute the lowest negative eigenvalue and corresponding eigenvector for a structural engineering buckling analysis, where the matrix A is the stiffness matrix K and the matrix B is the geometric stiffness matrix K_δ. The matrix K and M is positive semi-definite. A good guess at the eigenvector is available from static analysis and is stored in array USRSVC.

```
INTEGER*4 NFOUND, NDISCD, IER, WARNNG
REAL*8    USRSVC(NORDER)
CALL DSEVEX (1,'LOWEST','BUCKLING',.FALSE.,-1.0D30,
         .TRUE.,0.0D0,0.0D0,0,0.0D0,0.0D0,1,USRSVC,
         NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Example 3 Compute the 200 eigenvalues and corresponding eigenvectors closest to 3.14159 for a sparse matrix A. Less accuracy is needed than usual, but highly multiple eigenvalues are expected. It is known from previous analyses that eigenvalues closest to zero are about 1.5×10^{-5}.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVEX (200,'NEAREST','ORDINARY',.FALSE.,-1.0D30,
         .FALSE.,1.0D30,3.14159D0,10,1.0D-8,1.5D-5,
         0,0.0D0,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    check orthogonality with DSEVCK
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Name

DSEVRC
Return eigenvalue/eigenvector results

Purpose

This subprogram retrieves all or selected eigenvalues and eigenvectors after eigenextraction.

Usage

```
INTEGER*4      levalu, fstval, lstval, ldevct, ier
REAL*8         evalue(levalu), evectr(ldevct, levalu), global(150)
CALL DSEVRC(levalu, evalue, fstval, lstval, evectr, ldevct, global, ier)
```

Input

levalu Length of vector for storing eigenvalues.

fstval Ordinal position of first eigenvalue to be returned in **evalue**.

lstval Ordinal position of last eigenvalue to be returned in **evalue**.

The sign of **fstval** and **lstval** determine whether the eigenvalues returned are from the confirmed set or the discarded set. Positive indices indicate eigenvalues in the final confirmed trust region, while negative indices indicate the discardable set. The following pairs illustrate the meaning of **fstval** and **lstval**:

fstval	lstval	
1	**nfound**	All final trust-region eigenvalues.
−1	**−ndiscd**	All other eigenvalues.
1	**−ndiscd**	All computed eigenvalues.
i	*i*	*i*-th final trust-region eigenvalue.
−*i*	−*i*	*i*-th discardable eigenvalue.
i	*j*	*i*-th through *j*-th final trust-region eigenvalues.
i	−*j*	*i*-th through **nfound**-th final trust-region eigenvalues and 1st through *j*-th discardable eigenvalues.

ldevct The leading dimension of array **evectr** as declared in the calling program unit, with **ldevct** ≥ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

evalue List of selected eigenvalues.

evectr Corresponding array of eigenvectors. Each eigenvector is normalized such that

$x^T Bx = x^T Mx = 1$ (vibration)

$x^T Ax = x^T Kx = 1$ (buckling)

$x^T x = 1$ (ordinary)

ier Status response:

ier = 0	Normal return.
ier = –800	Incorrect processing path.
ier = –801	**levalu** not large enough.
ier = –802	**ldevct** smaller than order of problem.
ier = –803	System error.
ier = –804	Error in input, indices out of range.

Example 1 Retrieve all of the final trust-region eigenvalues and eigenvectors.

```
INTEGER*4 NEVALS,LDEVCT,NFOUND,IER
REAL*8    EVALUE(100),EVECTR(10000,100),GLOBAL(150)
NEVALS = 100
LDEVCT = 1000
CALL DSEVRC (NEVALS,EVALUE,1,NFOUND,EVECTR,LDEVCT,GLOBAL,
        IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Example 2 Retrieve all of the confirmed eigenvalues and the first three discarded eigenvalues, which lie in a cluster with the last confirmed eigenvalue and the corresponding eigenvectors.

```
INTEGER*4 NEVALS,IER
REAL*8    EVALUE(55),EVECTR(10000,100),GLOBAL(150)
NEVALS = 55
LDEVCT = 10000
CALL DSEVRC (NEVALS,EVALUE,1,-3,EVECTR,LDEVCT,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVRL
Return eigenvalue results

Purpose This subprogram retrieves all or selected eigenvalues after eigenextraction.

Usage

```
INTEGER*4      levalu, fstval, lstval, ier
REAL*8         evalue(levalu), global(150)
CALL DSEVRL(levalu, evalue, fstval, lstval, global, ier)
```

Input

levalu Length of vector for storing eigenvalues.

fstval Ordinal position of first eigenvalue to be returned in **evalue**.

lstval Ordinal position of last eigenvalue to be returned in **evalue**.

The sign of **fstval** and **lstval** determine whether eigenvalues returned are from the confirmed set or the discarded set. Positive indices indicate eigenvalues in the final confirmed trust region, while negative indices the discardable set. The following pairs illustrate the meaning of **fstval** and **lstval**:

fstval	**lstval**	
1	**nfound**	All final trust-region eigenvalues.
−1	**−ndiscd**	All other eigenvalues.
1	**−ndiscd**	All computed eigenvalues.
i	*i*	*i*-th final trust-region eigenvalue.
−*i*	−*i*	*i*-th discardable eigenvalue.
i	*j*	*i*-th through *j*-th final trust-region eigenvalues.
i	−*j*	*i*-th through **nfound**-th final trust-region eigenvalues and 1st through *j*-th discardable eigenvalues.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output	**evalue**	List of eigenvalues.	
	ier	Status response:	
		ier = 0	Normal return.
		ier = –800	Incorrect processing path.
		ier = –801	**levalu** not large enough.
		ier = –803	Input error, indices out of range.

Example 1 Retrieve all of the final trust region eigenvalues.

```
INTEGER*4 NEVALS,NFOUND,IER
REAL*8    EVALUE(NEVALS),GLOBAL(150)
NEVALS = 100
LDEVCT = 1000
CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Example 2 Retrieve all of the computed eigenvalues.

```
INTEGER*4 NEVALS,NDISCD,NFOUND,IER
REAL*8    EVALUE(NEVALS),GLOBAL(150)
IF ( NDISCD .GT. 0 ) THEN
    CALL DSEVRL (NEVALS,EVALUE,1,-NDISCD,GLOBAL,IER)
ELSE
    CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
ENDIF
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name

DSEVCK
Check accuracy of results

Purpose

This subprogram performs an a posteriori check on the accuracy of the computed eigenvalues and eigenvectors. The eigenvectors should satisfy the following pairs of equations:

$$X^TKX = \text{diag}(\{\lambda_i\}), \qquad X^TMX = I \qquad \text{(vibration)}$$

$$X^TK_\delta X = \text{diag}(\{1/\lambda_i\}), \qquad X^TKX = I \qquad \text{(buckling)}$$

$$X^TAX = \text{diag}(\{\lambda_i\}), \qquad X^TX = I \qquad \text{(ordinary)}$$

The subroutine computes the relevant matrix-matrix products to confirm that these equations hold to a reasonable numerical accuracy.

Usage

```
INTEGER*4      ier
LOGICAL*4      nodscd, ortwrn
REAL*8         discrp, global(150)
CALL DSEVCK(nodscd, ortwrn, discrp, global, ier)
```

Input

nodscd — If .TRUE., check only the eigenpairs whose eigenvalues are confirmed to lie in the single final trust region. If FALSE., check all of the computed eigenpairs, including those from 'discardable' eigenvalues.

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ortwrn — If .TRUE., the maximum relative discrepancy detected by checking the validity of these equations exceeded what was expected for the accuracy tolerance used in the eigenextraction.

discrp — Maximum relative discrepancy detected.

ier — Status response:

ier = 0	Normal return.
ier = –800	Incorrect processing path.
ier = –801	Error in storage allocation.
ier = –802	Error in retrieving eigenvectors.
ier = –804	System error.

Notes Additional diagnostic output from this subprogram is written onto the Fortran logical unit specified in the last call to DSEVIN or DSEVOC. The amount of output is independent of the message level value set in the last call to DSEVIN or DSEVOC.

Example Check only the final trust region eigenvalues and eigenvectors.

```
LOGICAL*4 ORTWRN
REAL*8    DISCRP,GLOBAL(150)
INTEGER*4 IER
CALL DSEVCK (.TRUE.,ORTWRN,DISCRP,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVDA
Deallocate working storage

Purpose This subprogram deallocates working storage at the end of processing. If the program using the package is going to continue execution when use of the package is completed, then it is recommended that the user deallocate the dynamically allocated working storage to reduce system impact.

Usage

```
INTEGER*4       ier
REAL*8          global(150)
CALL DSEVDA(global, ier)
```

Updated

global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier	Status response:	
	ier = 0	Normal return.
	ier = –801	Error in dynamic storage deallocation.

Example Deallocate working storage after use of package.

```
REAL*8 GLOBAL(150)
CALL DSEVDA (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVOC
Output control

Purpose This subprogram can be called at any point after subprogram DSEVIN to alter the output message level and the Fortran output unit number for message output.

Usage

```
INTEGER*4      msglvl, output
REAL*8         global(150)
CALL DSEVOC(msglvl, output, global)
```

Input

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages, summary statistics and inertia information.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output Fortran logical unit number to which all output will be written.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Example Increase message level from 1 to 2 while leaving the Fortran logical unit number for the output unit at 6.

```
REAL*8 GLOBAL(150)
CALL DSEVOC (2,6,GLOBAL)
```

Name DSEVPS
Print statistics

Purpose This subprogram prints available statistics about the original matrices, amount of work space in use, amount required for the next phase, and maximum amount used thus far. Also, this subprogram prints storage and arithmetic requirements, CPU time used, and computational rate for Cholesky factorization. The amount of information printed depends on the stage of execution. The number of lines of output ranges from 8 to 30, with the width of the lines being less than 80 characters.

Usage

REAL*8 global(150)
CALL DSEVPS(global)

Input

global Global communications array for this problem.

Example Print the statistics after the package has completed a numeric solution (either after DSEVSL or DSEVFS).

```
REAL*8 GLOBAL(150)
CALL DSEVPS (GLOBAL)
```

Name

DSEVRS
Restore problem state from a savefile

Purpose

This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSEVSV.

Usage

```
INTEGER*4       svfile, ier
REAL*8          global(150)
CALL DSEVRS(svfile, global, ier)
```

Input

svfile	Fortran logical unit number of a file containing the saved problem state as created by DSEVSV.

Output

global	Global communications array restored from **svfile**.	
ier	Status response:	
	ier = 0	Normal return.
	ier = –901	Error in storage allocation.
	ier = –902	I/O error detected by Fortran.

Example

Restart the eigenanalysis from the save file on Fortran logical unit 42 created by subroutine DSEVSV.

```
REAL*8 GLOBAL(150)
CALL DSEVRS (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name DSEVSV
Save problem state to a savefile

Purpose This subprogram saves the current problem for later restart at its current state. The global communication array and all working storage are written onto the user-specified I/O file using standard Fortran unformatted sequential write statements. The file is rewound before and after use.

Usage

```
INTEGER*4       svfile, ier
REAL*8          global(150)
CALL DSEVSV(svfile, global, ier)
```

Input

svfile Fortran logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –902	I/O error detected by Fortran.

Example Save the current state on Fortran logical unit 42.

```
REAL*8 GLOBAL(150)
CALL DSEVSV (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

6 Fast Fourier Transforms

Overview

This chapter explains how to use the VECLIB fast Fourier transform (FFT) subprograms. The operations covered are:

- One-, two-, and three-dimensional complex-to-complex FFT subprograms
- One-, two-, and three-dimensional real-to-complex FFT subprograms
- One-dimensional simultaneous complex-to-complex FFT subprograms
- One-dimensional simultaneous real-to-complex FFT subprograms

Chapter objectives

After reading this chapter you will:

- Understand VECLIB FFT subprogram restrictions
- Know how to augment subprograms with zero-value data points
- Know how to use the subprograms described in "Subprograms for Fast Fourier Transforms" on page 453

Associated documentation

The following documents provide supplemental material for this chapter:

Brigham, E.O. *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1974.

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Van Loan, C. *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: Society for Industrial and Applied Mathematics, 1992.

What you need to know to use these subprograms

Strictly speaking, an FFT is not a type of transform but a class of algorithms for efficiently computing the discrete Fourier transform (DFT).

Although the DFT is defined for any number of data points, the VECLIB FFT subprograms restrict the number of points to certain forms. The number of points in each direction must be a product of powers of 2, 3, and 5:

$$l_k = 2^{p_k}3^{q_k}5^{r_k}, \quad p_k, q_k, r_k \geq 0.$$

In addition, for real-to-complex and complex-to-real, DFTs l_1=1 or l_1 is even.

While this restriction may limit the use of the subprograms, the gain in speed is enormous. You can frequently adapt your data set to the VECLIB FFT subprograms by augmenting it with enough zero-value data points to reach the next acceptable number of points. Doing so slightly changes the problem, which may or may not be important. For example, adding zero-value points to a time series changes the implied sampling frequency, but adding zero-value points to data sets before using FFT subprograms to compute convolutions does not change the result.

Subprograms for Fast Fourier Transforms

The following sections describe FFT subprograms included with VECLIB.

Name C1DFFT/Z1DFFT
One-dimensional FFT

Purpose Given an array of complex data, these subprograms compute the one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S1DFFT and D1DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The one-dimensional forward DFT of $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

Finally, the one-dimensional unscaled inverse DFT of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms require that l be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r$$

where $p, q, r \geq 0$

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two **CALL** statements to the FFT subprogram using the same working storage array. Refer to "Example" on page 455

```
INTEGER*4       l, iopt, ier
COMPLEX*8       z(l)
REAL*4          work(5*l/2)
CALL C1DFFT(z, l, work, iopt, ier)

INTEGER*4       l, iopt, ier
COMPLEX*16      z(l)
REAL*8          work(5*l/2)
CALL Z1DFFT(z, l, work, iopt, ier)
```

Input

z Array of data to be transformed. Not used if **iopt** = –3.

l Number of data points, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.

iopt Option flag:

iopt = +1	Compute forward transform.
iopt = –1	Compute scaled inverse transform.
iopt = –2	Compute unscaled inverse transform.
iopt = –3	Initialize **work** for subsequent transforms of length **l**.

Working Storage

work If **iopt** = –3, **work** is initialized for computing transforms of length **l**.

If **iopt** ≠ –3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was –3.

Output

z If **iopt** ≠ –3, transformed data replaces the input if **ier** = 0 is returned. Not used as output if **iopt** = –3.

ier	Status response:	
	ier = 0	Normal return—transform or initialization successful.
	ier = −1	**l** < 0.
	ier = −2	**l** not of the required form.
	ier = −3	Invalid value of **iopt**.
	ier = −4	Insufficient dynamic memory available for workspace.

Example Compute the forward discrete Fourier transform of two COMPLEX*8 data sets of length 1024. Here, length of working storage is 5 × 1024/2 = 2560.

```
INTEGER*4 L,IOPT,IER
COMPLEX*8 Z1(1024),Z2(1024)
REAL*4    WORK(5*1024/2)
L = 1024
IOPT = -3
CALL C1DFFT (Z1,L,WORK,IOPT,IER)       !  INITIALIZE
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM C1DFFT.'
    STOP
END IF
CALL C1DFFT (Z1,L,WORK,IOPT,IER)       !  FIRST TRANSFORM
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM C1DFFT.'
    STOP
END IF
CALL C1DFFT (Z2,L,WORK,IOPT,IER)       !  SECOND TRANSFORM
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM C1DFFT.'
    STOP
END IF
```

Name S1DFFT/D1DFFT
One-dimensional FFT

Purpose Given a set of complex data with the real and imaginary parts in separate real arrays, these subprograms compute the one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C1DFFT and Z1DFFT, perform the same operation but with the complex data presented in a complex array.

The one-dimensional forward discrete Fourier transform of $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

Finally, the one-dimensional unscaled inverse DFT of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms require that l be a product of power of 2, 3 and 5, that is, of the form

$$l = 2^p 3^q 5^r$$

with $p, q, r \geq 0$

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two **CALL** statements to the FFT subprogram using the same working storage array. Refer to "Example" on page 458.

```
INTEGER*4       l, iopt, ier
REAL*4          x(l), y(l), work(5*l/2)
CALL S1DFFT(x, y, l, work, iopt, ier)

INTEGER*4       l, iopt, ier
REAL*8          x(l), y(l), work(5*l/2)
CALL D1DFFT(x, y, l, work, iopt, ier)
```

Input

x Array of real parts of the data to be transformed. Not used if **iopt** = −3.

y Array of imaginary parts of the data to be transformed. Not used if **iopt** = −3.

l Number of data points, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.

iopt Option flag:

iopt = +1	Compute forward transform.
iopt = −1	Compute scaled inverse transform.
iopt = −2	Compute unscaled inverse transform.
iopt = −3	Initialize **work** for subsequent transforms of length **l**.

Working Storage

work If **iopt** = −3, **work** is initialized for computing transforms of length **l**.

If **iopt** ≠ −3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was −3.

Output

x and y If **iopt** ≠ −3, the transformed data replaces the input if **ier** = 0 is returned. Not used as output if **iopt** = −3.

ier Status response:

ier = 0	Normal return—transform or initialization successful.
ier = −1	**l** < 0.
ier = −2	**l** not of the required form.
ier = −3	Invalid value of **iopt**.
ier = −4	Insufficient dynamic memory available for workspace.

Example Compute the forward discrete Fourier transform of two REAL*8 data sets of length 512. Here, the length of the working storage is 5 × 512/2 = 1280.

```
INTEGER*4 L,IOPT,IER
REAL*8    X1(512),Y1(512),X2(512),Y2(512),WORK(5*512/2)
L = 512
IOPT = -3
CALL D1DFFT (X1,Y1,L,WORK,IOPT,IER)    !  INITIALIZE
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM D1DFFT.'
    STOP
END IF
IOPT = 1
CALL D1DFFT (X1,Y1,L,WORK,IOPT,IER)    !  FIRST TRANSFORM
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM D1DFFT.'
    STOP
END IF
CALL D1DFFT (X2,Y2,L,WORK,IOPT,IER)    !  SECOND TRANSFORM
IF ( IER .NE. 0 ) THEN
    PRINT *,'IER =',IER,' FROM D1DFFT.'
    STOP
END IF
```

Name

C2DFFT/Z2DFFT
Two-dimensional FFT

Purpose

Given an array of complex data, these subprograms compute the two-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S2DFFT and D2DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The two-dimensional forward discrete Fourier transform of $z(n_1,n_2)$, for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \ldots, l_1, m_2 = 1, 2, \ldots, l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse discrete Fourier transform of $Z(m_1,m_2)$, for $m_1 = 1, 2, \ldots, l_1$ and $m_2 = 1, 2, \ldots, l_2$, is defined by

$$z(n_1, n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k,q_k,r_k \geq 0$, k=1,2.

Usage

```
INTEGER*4       l1, l2, ldz, iopt, ier
COMPLEX*8       z(ldz, l2)
CALL C2DFFT(z, l1, l2, ldz, iopt, ier)

INTEGER*4       l1, l2, ldz, iopt, ier
COMPLEX*16      z(ldz, l2)
CALL Z2DFFT(z, l1, l2, ldz, iopt, ier)
```

Input

z	Array of data to be transformed.
l1	Number of rows of data, of the form **l1** = $2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.

	l2	Number of columns of data, of the form $\mathbf{l2} = 2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.	
	ldz	The leading dimension of array **z**, with **ldz** ≥ **l1**.	
	iopt	Option flag:	
		iopt ≥ 0	Compute forward transform.
		iopt < 0	Compute inverse transform.
Output	**z**	The transformed data replaces the input if **ier** = 0 is returned.	
	ier	Status response:	
		ier = 0	Normal return—transform successful.
		ier = −1	**l1** not of the required form.
		ier = −2	**l2** not of the required form.
		ier = −3	**ldz** < **l1**.
		ier ≤ −4	Probable error in **ldz** or dimensions of **z**.

Name

S2DFFT/D2DFFT
Two-dimensional FFT

Purpose

Given a set of complex data with the real and imaginary parts in separate real arrays, these subprograms compute the two-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C2DFFT and Z2DFFT, perform the same operation but with the complex data presented in a complex array.

The two-dimensional forward DFT of $z(n_1,n_2)$, for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$, is defined by

$$Z(m_1,m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1,n_2)\, e^{-2\pi i(m_1-1)(n_1-1)/l_1}\, e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse DFT of $Z(m_1,m_2)$, for $m_1 = 1, 2, \ldots, l_1$ and $m_2 = 1, 2, \ldots, l_2$, is defined by

$$z(n_1,n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1,m_2)\, e^{+2\pi i(m_1-1)(n_1-1)/l_1}\, e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$.

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k}\, 3^{q_k}\, 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2$.

Usage

```
INTEGER*4       l1, l2, ldxy, iopt, ier
REAL*4          x(ldxy, l2), y(ldxy, l2)
CALL S2DFFT(x, y, l1, l2, ldxy, iopt, ier)

INTEGER*4       l1, l2, ldxy, iopt, ier
REAL*8          x(ldxy, l2), y(ldxy, l2)
CALL D2DFFT(x, y, l1, l2, ldxy, iopt, ier)
```

Input	**x**	Array of real parts of the data to be transformed.	
	y	Array of imaginary parts of the data to be transformed.	
	l1	Number of rows of data, of the form $\mathbf{l1} = 2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.	
	l2	Number of columns of data, of the form $\mathbf{l2} = 2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.	
	ldxy	The leading dimension of arrays **x** and **y**, with **ldxy** ≥ **l1**.	
	iopt	Option flag:	
		iopt ≥ 0	Compute forward transform.
		iopt < 0	Compute inverse transform.
Output	**x and y**	The transformed data replaces the input if **ier** = 0 is returned.	
	ier	Status response:	
		ier = 0	Normal return—transform successful.
		ier = −1	**l1** not of the required form.
		ier = −2	**l2** not of the required form.
		ier = −3	**ldxy** < **l1**.
		ier ≤ −4	Probable error in **ldxy** or dimensions of **x** and **y**.

Name

C3DFFT/Z3DFFT
Three-dimensional FFT

Purpose

Given an array of complex data, these subprograms compute the three-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S3DFFT and D3DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The three-dimensional forward DFT of $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, $m_3 = 1, 2, ..., l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse DFT of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $m_3 = 1, 2, ..., l_3$, is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$.

These subprograms require that l_1, l_2, and l_3 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1,2,3$.

Usage

```
INTEGER*4       l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*8       z(ldz, mdz, l3)
CALL C3DFFT(z, l1, l2, l3, ldz, mdz, iopt, ier)

INTEGER*4       l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*16      z(ldz, mdz, l3)
CALL Z3DFFT(z, l1, l2, l3, ldz, mdz, iopt, ier)
```

Input	**z**	Array of data to be transformed.
	l1	Number of rows of data, of the form **l1** = $2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
	l2	Number of columns of data, of the form **l2** = $2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	l3	Number of planes of data, of the form **l3** = $2^{p_3}\ 3^{q_3}\ 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
	ldz	The leading dimension of array **z**, with **ldz** ≥ **l1**.
	mdz	The middle dimension of array **z**, with **mdz** ≥ **l2**.
	iopt	Option flag:
		iopt ≥ 0 Compute forward transform.
		iopt < 0 Compute inverse transform.
Output	**z**	The transformed data replaces the input if **ier** = 0 is returned.
	ier	Status response:
		ier = 0 Normal return—transform successful.
		ier = –1 **l1** not of the required form.
		ier = –2 **l2** not of the required form.
		ier = –3 **l3** not of the required form.
		ier = –4 **ldz** < **l1**.
		ier = –5 **mdz** < **l2**.
		ier = –6 Probable error in **ldz** or **mdz**.

Name S3DFFT/D3DFFT
Three-dimensional FFT

Purpose Given a set of complex data with real and imaginary parts in separate real arrays, these subprograms compute the three-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C3DFFT and Z3DFFT, perform the same operation but with the complex data presented in a complex array.

The three-dimensional forward DFT of $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, $m_3 = 1, 2, \ldots, l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse DFT of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $m_3 = 1, 2, \ldots, l_3$, is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$.

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

These subprograms require that l_1, l_2, and l_3 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k}\ 3^{q_k}\ 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2, 3$.

Usage

```
INTEGER*4      l1, l2, l3, ldxy, mdxy, iopt, ier
REAL*4         x(ldxy, mdxy, l3), y(ldxy, mdxy, l3)
CALL S3DFFT(x, y, l1, l2, l3, ldxy, mdxy, iopt, ier)

INTEGER*4      l1, l2, l3, ldxy, mdxy, iopt, ier
REAL*8         x(ldxy, mdxy, l3), y(ldxy, mdxy, l3)
CALL D3DFFT(x, y, l1, l2, l3, ldxy, mdxy, iopt, ier)
```

Input

x	Array of real parts of the data to be transformed.
y	Array of imaginary parts of the data to be transformed.
l1	Number of rows of data, of the form $\mathbf{l1} = 2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
l2	Number of columns of data, of the form $\mathbf{l2} = 2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
l3	Number of planes of data, of the form $\mathbf{l3} = 2^{p_3}\ 3^{q_3}\ 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
ldxy	The leading dimension of arrays **x** and **y**, with **ldxy** ≥ **l1**.
mdxy	The middle dimension of arrays **x** and **y**, with **mdxy** ≥ **l2**.
iopt	Option flag:
	iopt ≥ 0 — Compute forward transform.
	iopt < 0 — Compute inverse transform.

Output

x and y	The transformed data replaces the input if **ier** = 0 is returned.
ier	Status response:
	ier = 0 — Normal return—transform successful.
	ier = −1 — **l1** not of the required form.
	ier = −2 — **l2** not of the required form.
	ier = −3 — **l3** not of the required form.
	ier = −4 — **ldxy** < **l1**.
	ier = −5 — **mdxy** < **l2**.
	ier = −6 — Probable error in **ldxy** or **mdxy**.

Name

CFFTS/ZFFTS
Simultaneous one-dimensional FFT

Purpose

Given a number of sets of one-dimensional complex data in a complex array, these subprograms compute all of their one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, SFFTS and DFFTS, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one transform.

The one-dimensional forward DFT of a complex data set $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of a data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z((m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms perform forward or inverse transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$.

Usage

```
INTEGER*4       l, incl, n, incn, iopt, ier
COMPLEX*8       z(lenz)
CALL CFFTS(z, l, incl, n, incn, iopt, ier)

INTEGER*4       l, incl, n, incn, iopt, ier
COMPLEX*16      z(lenz)
CALL ZFFTS(z, l, incl, n, incn, iopt, ier)
```

Input

z Array containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **z** will be a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages. Treating **z** as a one-dimensional array results in

$$\mathbf{lenz} = (\mathbf{l}-1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

The i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{z}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1).$$

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $p, q, r \ge 0$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in array **z**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0	Compute forward transform.
iopt < 0	Compute inverse transform.

Output

z The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.

ier Status response:

ier = 0	Normal return—transform successful.
ier = –1	**l** not of the form $2^p\ 3^q\ 5^r$ for $p, q, r \ge 0$.
ier = –2	**incl** ≤ 0.
ier = –3	**n** ≤ 0.
ier = –4	**incn** ≤ 0.
ier = –5	**l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **z** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, that is, all but one subscript will be constant within a data set.

If **z** is a two-dimensional array of dimension **ldz** by **mdz**, and if the data sets are stored in the columns of **z**, **l** ≤ **ldz**, **n** ≤ **mdz**, **incl** = 1, and **incn** = **ldz**. For example,

CALL CFFTS (z, l, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and data sets are stored in rows of **z**, **l** ≤ **mdz**, **n** ≤ **ldz**, **incl** = **ldz**, and **incn** = 1. For example,

CALL CFFTS (z, l, ldz, n, 1, iopt, ier)

If **z** is a three-dimensional array of dimension **ldz** by **mdz** by **ndz**, then **incl** and **incn** will usually be 1, **ldz**, or **ldz** × **mdz**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st	subscript, use	**incl** = 1.
2nd	subscript, use	**incl** = **ldz**.
3rd	subscript, use	**incl** = **ldz** × **mdz**.

Similarly, if the subscript that varies between data sets is the

1st	subscript, use	**incn** = 1.
2nd	subscript, use	**incn** = **ldz**.
3rd	subscript, use	**incn** = **ldz** × **mdz**.

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same element of **z**. These subprograms detect this situation and return **ier** = −5 if

$$\textbf{incl} < \textbf{n} \times \gcd(\textbf{incl}, \textbf{incn})$$

and

$$\textbf{incn} < \textbf{l} \times \gcd(\textbf{incl}, \textbf{incn})$$

where gcd(.,.) is the greatest common divisor.

Example 1 Compute the forward DFT of 256 COMPLEX*8 data sets of length 1024. The data sets are stored as the columns of array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL CFFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Example 2 Compute the inverse DFT of 1024 COMPLEX*8 data sets of length 256. The data sets are stored as the rows of array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL CFFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Name

SFFTS/DFFTS
Simultaneous one-dimensional FFT

Purpose

Given a number of sets of one-dimensional complex data with real and imaginary parts in separate real arrays, these subprograms compute all of their one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, CFFTS and ZFFTS, perform the same operation but with the complex data presented in a complex array. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one transform.

The one-dimensional forward discrete Fourier transform of a complex set of data $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

These subprograms perform forward or inverse transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r$$

where $p, q, r \geq 0$.

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

Usage

```
INTEGER*4       l, incl, n, incn, iopt, ier
REAL*4          x(lenxy), y(lenxy)
CALL SFFTS(x, y, l, incl, n, incn, iopt, ier)

INTEGER*4       l, incl, n, incn, iopt, ier
REAL*8          x(lenxy), y(lenxy)
CALL DFFTS(x, y, l, incl, n, incn, iopt, ier)
```

Input

x and y Arrays containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **x** and **y** will be two- or three-dimensional arrays with each data set being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **x** and **y** as one-dimensional arrays results in

$$\mathbf{lenxy} = (\mathbf{l}-1)\times\mathbf{incl}+(\mathbf{n}-1)\times\mathbf{incn}+1.$$

The real and imaginary parts of the i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, are stored in

$$\mathbf{x}((i-1)\times\mathbf{incl}+(j-1)\times\mathbf{incn}+1)$$

and

$$\mathbf{y}((i-1)\times\mathbf{incl}+(j-1)\times\mathbf{incn}+1),$$

respectively.

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $p, q, r \ge 0$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in arrays **x** and **y**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0	Compute forward transform.
iopt < 0	Compute inverse transform.

Output

x and y	The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.	
ier	Status response:	
	ier = 0	Normal return—transform successful.
	ier = –1	**l** not of the form $2^p\ 3^q\ 5^r$ for $p, q, r \geq 0$.
	ier = –2	**incl** ≤ 0.
	ier = –3	**n** ≤ 0.
	ier = –4	**incn** ≤ 0.
	ier = –5	**l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **x** and **y** will be two- or three-dimensional arrays with each data set being a one-dimensional section of the arrays, that is, all but one subscript will be constant within a data set.

If **x** and **y** are two-dimensional arrays of dimension **ldxy** by **mdxy**, and if the data sets are stored in the columns of **x** and **y**, **l** ≤ **ldxy**, **n** ≤ **mdxy**, **incl** = 1, and **incn** = **ldxy**. For example:

CALL SFFTS (x, y, l, 1, n, ldxy, iopt, ier)

If **x** and **y** are two-dimensional arrays as above and data sets are stored in rows of **x** and **y**, **l** ≤ **mdxy**, **n** ≤ **ldxy**, **incl** = **ldxy**, and **incn** = 1. For example:

CALL SFFTS (x, y, l, ldxy, n, 1, iopt, ier)

If **x** and **y** are three-dimensional arrays of dimension **ldxy** by **mdxy** by **ndxy**, then **incl** and **incn** will usually be 1, **ldxy**, or **ldxy**×**mdxy**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st	subscript, use	**incl** = 1.
2nd	subscript, use	**incl** = **ldxy**.
3rd	subscript, use	**incl** = **ldxy**×**mdxy**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldxy**.
3rd subscript, use **incn** = **ldxy**×**mdxy**.

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same elements of **x** and **y**. These subprograms detect this situation and return **ier** = −5 if

$$\mathbf{incl} < \mathbf{n} \times \gcd(\mathbf{incl}, \mathbf{incn})$$

and

$$\mathbf{incn} < \mathbf{l} \times \gcd(\mathbf{incl}, \mathbf{incn})$$

where gcd(.,.) is the greatest common divisor.

Example 1 Compute the forward discrete Fourier transform of 256 complex data sets of length 1024. Real and imaginary parts of data sets are stored as columns of arrays X and Y whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
REAL*4    X(1025,256),Y(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL SFFTS (X,Y,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Example 2 Compute the inverse discrete Fourier transform of 1024 complex data sets of length 256. Real and imaginary parts of data sets are stored as rows of arrays X and Y whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
REAL*4    X(1025,256),Y(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL SFFTS (X,Y,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Name CRC1FT/ZRC1FT
Real-to-complex one-dimensional FFT

Purpose These subprograms compute either the forward real-to-complex or the inverse complex-to-real, one-dimensional, DFT using a radix 2-3-5 FFT algorithm optimized for real input or output, stored in a complex array. Two companion subprograms, SRC1FT and DRC1FT, perform a similar operation but in a space-conserving manner.

The one-dimensional forward DFT of a real data set, $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, that is,

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2 + 1 + m) = \bar{Z}(l/2 + 1 - m), m = 1, 2, ..., l/2 - 1$$

where $\bar{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the one-dimensional real scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

Finally, the one-dimensional, real, unscaled, inverse DFT of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms require that l be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$ and where either $l = 1$ or l is even.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. Therefore, you will always have at least two **CALL** statements to the FFT subprogram using the same working storage array.

```
INTEGER*4       l, iopt, ier
COMPLEX*8       z(l)
REAL*4          work(2*l)
CALL CRC1FT(z, l, work, iopt, ier)

INTEGER*4       l, iopt, ier
COMPLEX*16      z(l)
REAL*8          work(2*l)
CALL ZRC1FT(z, l, work, iopt, ier)
```

Input

z Array of data to be transformed. For a forward real-to-complex transform, only the real parts of **z** are used. For an inverse complex-to-real transform, only the first l/2+1 elements are used. Not used at all if **iopt** = –3.

l Number of data points, of the form $l = 2^p 3^q 5^r$, where $p, q, r \geq 0$ and where either **l** = 1 or **l** is even.

iopt Option flag:

iopt = +1	Compute forward real-to-complex transform.
iopt = –1	Compute scaled inverse complex-to-real transform.
iopt = –2	Compute unscaled inverse complex-to-real transform.
iopt = –3	Initialize **work** for subsequent transforms of length **l**.

Working Storage

work If **iopt** = –3, **work** is initialized for computing transforms of length **l**.

If **iopt** ≠ –3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was –3.

Output

z

If **iopt** ≠ −3, the transformed data replaces the input if **ier** = 0 is returned.

For a forward real-to-complex transform, all **l** elements of the transform are returned.

For an inverse complex-to-real transform, the real result is stored in the real parts of **z** and the imaginary parts of **z** are set to zero.

Not used as output if **iopt** = −3.

ier

Status response:

ier = 0	Normal return—transform or initialization successful.
ier = −1	**l** < 0.
ier = −2	**l** not of the required form.
ier = −3	Invalid value of **iopt**.
ier = −4	Insufficient dynamic memory available for workspace.

Name SRC1FT/DRC1FT
Real-to-complex one-dimensional FFT

Purpose Given a set of real data in a real array, these subprograms compute the nonredundant part of the complex one-dimensional forward DFT using a radix 2-3-5 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric complex one-dimensional data set, these subprograms compute the real inverse discrete Fourier transform. A pair of companion subprograms, CRC1FT and ZRC1FT, performs a similar operation, but with the real and complex data presented in a complex array.

The one-dimensional forward DFT of a real data set, $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, that is,

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2 + 1 + m) = \bar{Z}(l/2 + 1 - m), m = 1, 2, ..., l/2 - 1,$$

where $\bar{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the one-dimensional real scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

Finally, the one-dimensional real unscaled inverse DFT of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\ e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms require that l be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$ and where either $l = 1$ or l is even.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two call statements to the FFT subprogram, using the same working storage array.vb

```
INTEGER*4       l, iopt, ier
REAL*4          x(l+2), work(2*l)
CALL SRC1FT(x, l, work, iopt, ier)

INTEGER*4       l, iopt, ier
REAL*8          x(l+2), work(2*l)
CALL DRC1FT(x, l, work, iopt, ier)
```

Input

x Array containing **l** real data points or the first **l**/2+1 complex data points of a conjugate-symmetric complex data set of length **l**, to be transformed.

For a forward real-to-complex transform, the i-th real data point is stored in **x**(i), $i = 1, 2, ..., \mathbf{l}$.

For an inverse complex-to-real transform, the real part of the i-th complex data point is stored in **x**($2\times i-1$) and the imaginary part of the i-th data point is stored in **x**($2\times i$), $i = 1, 2, ..., \mathbf{l}/2+1$.

Not used if **iopt** = –3.

l Number of data points, of the form $\mathbf{l} = 2^p 3^q 5^r$, where $p, q, r \geq 0$ and where either **l** = 1 or **l** is even.

iopt Option flag:

iopt = +1	Compute forward transform.
iopt = –1	Compute scaled inverse transform.
iopt = –2	Compute unscaled inverse transform.
iopt = –3	Initialize **work** for subsequent transforms of length **l**.

Working Storage	**work**	If **iopt** = –3, **work** is initialized for computing transforms of length **l**. If **iopt** ≠ –3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was –3.
Output	**x**	If **iopt** ≠ –3, the transformed data replaces the input if **ier** = 0 is returned. For a forward real-to-complex transform, the real part of the i-th complex output point is stored in **x**$(2\times i-1)$ and the imaginary part of the i-th output point is stored in **x**$(2\times i)$, i = 1, 2, ..., **l**/2+1. If needed, the remaining **l**/2 – 1 complex output values may be formed by using the conjugate symmetry condition. For an inverse complex-to-real transform, the i-th real output point is stored in **x**(i), i = 1, 2, ..., **l**. Not used as output if **iopt** = –3.
	ier	Status response: **ier** = 0 — Normal return—transform or initialization successful. **ier** = –1 — **l** < 0. **ier** = –2 — **l** not of the required form. **ier** = –3 — Invalid value of **iopt**. **ier** = –4 — Insufficient dynamic memory available for workspace.

Name CRC2FT/ZRC2FT
Real-to-complex two-dimensional FFT

Purpose These subprograms compute either the forward real-to-complex or the inverse complex-to-real, two-dimensional, discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm optimized for real input or output, stored in a complex array. A pair of companion subprograms, SRC2FT and DRC2FT, performs a similar operation, but in a space-conserving manner.

The two-dimensional, complex, forward discrete Fourier transform of a real data set, $z(n_1,n_2)$, for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2)$ satisfy the two-dimensional, conjugate-symmetry conditions:

$$Im(Z(1,1)) = 0,$$

$$Z(m_1, 1) = \overline{Z}(l_1 + 2 - m_1, 1), m_1 = 2, 3, \ldots, l_1,$$

$$Z(1, m_2) = \overline{Z}(1, l_2 + 2 - m_2), m_2 = 2, 3, \ldots, l_2,$$

and

$$Z(m_1, m_2) = \overline{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2), m_k = 2, 3, \ldots, l_k, k = 1, 2,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m_1,m_2)$, for $m_1 = 1, 2, \ldots, l_1$ and $m_2 = 1, 2, \ldots, l_2$, is a conjugate-symmetric complex data set, the two-dimensional, real, scaled, inverse discrete Fourier transform of $Z(m_1,m_2)$ is defined by

$$z(n_1, n_2) = \frac{l}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2$, and where either $l_1 = 1$ or l_1 is even.

Usage

```
INTEGER*4      l1, l2, ldz, iopt, ier
COMPLEX*8      z(ldz, l2)
CALL CRC2FT(z, l1, l2, ldz, iopt, ier)

INTEGER*4      l1, l2, ldz, iopt, ier
COMPLEX*16     z(ldz, l2)
CALL ZRC2FT(z, l1, l2, ldz, iopt, ier)
```

Input

z Array of data to be transformed. For a forward real-to-complex transform, only the real parts of **z** are used as input. For an inverse complex-to-real transform, only the first **l1**/2+1rows of **z** are used as input.

l1 Number of rows of data, of the form $\mathbf{l1} = 2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $q_1, r_1 \geq 0$ and either **l1** = 1 or **l1** is even.

l2 Number of columns of data, of the form $\mathbf{l2} = 2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.

ldz The leading dimension of array **z**, with **ldz** ≥ **l1**.

iopt Option flag:

iopt ≥ 0	Compute forward transform.
iopt < 0	Compute inverse transform.

Output

z The transformed data replaces the input if **ier** = 0 is returned. For an inverse complex-to-real transform, the real result is stored in the real parts of **z** and the imaginary parts of **z** are set to zero.

ier Status response:

ier = 0	Normal return—transform successful.
ier = −1	**l1** not of the required form.
ier = −2	**l2** not of the required form.
ier = −3	**ldz** < **l1**.
ier = −4	Probable error in **ldz** or dimensions of **z**.

Name SRC2FT/DRC2FT
Real-to-complex two-dimensional FFT

Purpose Given a set of two-dimensional real data, these subprograms compute the nonredundant portion of the complex, two-dimensional, forward DFT using a radix 2-3-5 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric two-dimensional complex data set, these subprograms compute the real inverse discrete Fourier transform using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRC2FT and ZRC2FT, performs similar operations, but with the real or complex data presented in a complex array. The companion subprograms require more storage than the ones described here.

The two-dimensional complex forward DFT of a real data set, $z(n_1,n_2)$, for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2)$ satisfy the two-dimensional conjugate-symmetry conditions:

$$Im(Z(1, 1)) = 0,$$

$$Z(m_1, 1) = \bar{Z}(l_1 + 2 - m_1, 1), m_1 = 2, 3, \ldots, l_1,$$

$$Z(1, m_2) = \bar{Z}(1, l_2 + 2 - m_1), m_2 = 2, 3, \ldots, l_2,$$

and

$$Z(m_1, m_2) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2), m_k = 2, 3, \ldots, l_k, k = 1, 2,$$

where $\bar{Z}$ is the complex conjugate of Z.

Alternatively, the two-dimensional, real, inverse DFT of $Z(m_1,m_2)$, for $m_1 = 1, 2, \ldots, l_1$ and $m_2 = 1, 2, \ldots, l_2$, is defined by

$$z(n_1, n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k}3^{q_k}5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2$, and where either l_1=1 or l_1 is even.

Usage

```
INTEGER*4       l1, l2, ldx, iopt, ier
REAL*4          x(ldx, l2)
CALL SRC2FT(x, l1, l2, ldx, iopt, ier)

INTEGER*4       l1, l2, ldx, iopt, ier
REAL*8          x(ldx, l2)
CALL DRC2FT(x, l1, l2, ldx, iopt, ier)
```

Input

x Array of data to be transformed.

For a forward real-to-complex transform, the real data point $z(n_1,n_2)$ is stored in $\mathbf{x}(n_1,n_2)$, $n_1 = 1, 2, ..., \mathbf{l1}$, $n_2 = 1, 2, ..., \mathbf{l2}$.

For an inverse complex-to-real transform, the real part of $Z(m_1,m_2)$ is stored in $\mathbf{x}(2\times m_1-1,m_2)$and the imaginary part is stored in $\mathbf{x}(2\times m_1,m_2)$, $m_1 = 1, 2, ..., \mathbf{l1}/2+1, m_2 = 1, 2, ..., \mathbf{l2}$.

l1 Number of rows of data, of the form $\mathbf{l1} = 2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $q_1, r_1 \geq 0$ and either **l1** = 1 or **l1** is even.

l2 Number of columns of data, of the form $\mathbf{l2} = 2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.

ldx The leading dimension of array **x**, with **ldx** ≥ **l1**+2.

iopt Option flag:

iopt ≥ 0	Compute forward transform.
iopt < 0	Compute inverse transform.

Output

x

The transformed data replaces the input if **ier** = 0 is returned.

For a forward real-to-complex transform, the real part of $Z(m_1,m_2)$ is stored in $\mathbf{x}(2\times m_1-1,m_2)$and the imaginary part is stored in $\mathbf{x}(2\times m_1,m_2)$, $m_1 = 1, 2, ..., \mathbf{l1}/2+1, m_2 = 1, 2, ..., \mathbf{l2}$. If needed, the remaining (**l1**/2 − 1) × **l2** complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the real output point $z(n_1,n_2)$ is stored in $\mathbf{x}(n_1,n_2)$, $n_1 = 1, 2, ..., \mathbf{l1}, n_2 = 1, 2, ..., \mathbf{l2}$.

ier

Status response:

ier = 0	Normal return—transform successful.
ier = −1	**l1** not of the required form.
ier = −2	**l2** not of the required form.
ier = −3	**ldx** < **l1**+2.
ier = −4	Probable error in **ldx** or dimensions of **x**.

Name

CRC3FT/ZRC3FT
Real-to-complex three-dimensional FFT

Purpose

These subprograms compute either the forward real-to-complex or the inverse complex-to-real, three-dimensional DFT using a radix 2-3-5 FFT algorithm optimized for real input or output, stored in a complex array. A pair of companion subprograms, SRC3FT and DRC3FT, performs a similar operation but in a space-conserving manner.

The three-dimensional complex forward DFT of a real data set $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, $m_3 = 1, 2, \ldots, l_3$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2,m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$Im(Z(1, 1, 1)) = 0,$$

$$Z(m_1, 1, 1) = \overline{Z}(l_1 + 2 - m_1, 1, 1), m_1 = 2, 3, \ldots, l_1,$$

$$Z(1, m_2, 1) = \overline{Z}(1, l_2 + 2 - m_2, 1), m_2 = 2, 3, \ldots, l_2,$$

$$Z(1, 1, m_3) = \overline{Z}(1, 1, l_3 + 2 - m_3), m_3 = 2, 3, \ldots, l_3,$$

$$Z(m_1, m_2, 1) = \overline{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, 1), m_k = 2, 3, \ldots, l_k, k = 1, 2,$$

$$Z(m_1, 1, m_3) = \overline{Z}(l_1 + 2 - m_1, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 1, 3,$$

$$Z(1, m_2, m_3) = \overline{Z}(1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 2, 3,$$

and

$$Z(m_1, m_2, m_3) = \overline{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 1, 2, 3,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $m_3 = 1, 2, ..., l_3$, is a conjugate-symmetric complex data set, the three-dimensional, real, scaled, inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$ is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/}$$

for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$.

These subprograms require that l_1, l_2, and l_3 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2, 3$, and where either $l_1 = 1$ or l_1 is even.

Usage

```
INTEGER*4       l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*8       z(ldz, mdz, l3)
CALL CRC3FT(z, l1, l2, l3, ldz, mdz, iopt, ier)

INTEGER*4       l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*16      z(ldz, mdz, l3)
CALL ZRC3FT(z, l1, l2, l3, ldz, mdz, iopt, ier)
```

Input

z	Array of data to be transformed. For a forward real-to-complex transform, only the real parts of **z** are used as input. For an inverse complex-to-real transform, only the first **l1**/2+1 rows of **z** are used as input.
l1	Number of rows of data, of the form **l1** = $2^{p_1}\ 3^{q_1}\ 5^{r_1}$, with $q_1, r_1 \geq 0$ and either **l1** = 1 or $p_1 \geq 1$.
l2	Number of columns of data, of the form **l2** = $2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
l3	Number of planes of data, of the form **l3** = $2^{p_3}\ 3^{q_3}\ 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.

	ldz	The leading dimension of array **z**, with **ldz** ≥ **l1**.	
	mdz	The middle dimension of array **z**, with **mdz** ≥ **l2**.	
	iopt	Option flag:	
		iopt ≥ 0	Compute forward transform.
		iopt < 0	Compute inverse transform.
Output	**z**	The transformed data replaces the input if **ier** = 0 is returned. For an inverse complex-to-real transform, the real result is stored in the real parts of **z** and the imaginary parts of **z** are set to zero.	
	ier	Status response:	
		ier = 0	Normal return—transform successful.
		ier = –1	**l1** not of the required form.
		ier = –2	**l2** not of the required form.
		ier = –3	**l3** not of the required form.
		ier = –4	**ldz** < **l1**.
		ier = –5	**mdz** < **l2**.
		ier ≤ –6	Probable error in **ldx** or **mdx**.

Name SRC3FT/DRC3FT
Real-to-complex three-dimensional FFT

Purpose Given a set of three-dimensional real data, these subprograms compute the nonredundant portion of the complex three-dimensional forward DFT using a radix 2-3-5 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric, three-dimensional, complex data set, these subprograms compute the real inverse DFT using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRC3FT and ZRC3FT, performs similar operations, but with the real or complex data presented in a complex array. These companion subprograms require more storage than the ones described here.

The three-dimensional, complex, forward a DFT of a real data set $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, $m_3 = 1, 2, \ldots, l_3$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2,m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$Im(Z(1, 1, 1)) = 0,$$

$$Z(m_1, 1, 1) = \overline{Z}(l_1 + 2 - m_1, 1, 1), m_1 = 2, 3, \ldots, l_1,$$

$$Z(1, m_2, 1) = \overline{Z}(1, l_2 + 2 - m_1, 1), m_2 = 2, 3, \ldots, l_2,$$

$$Z(1, 1, m_3) = \overline{Z}(1, 1, l_3 + 2 - m_1), m_3 = 2, 3, \ldots, l_3,$$

$$Z(m_1, m_2, 1) = \overline{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, 1), m_k = 2, 3, \ldots, l_k, k = 1, 2,$$

$$Z(m_1, 1, m_3) = \overline{Z}(l_1 + 2 - m_1, 1, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 1, 3,$$

$$Z(1, m_2, m_3) = \overline{Z}(1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 2, 3,$$

and

$$Z(m_1, m_2, m_3) = \overline{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \ldots, l_k, k = 1, 2, 3,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, the three-dimensional, real, inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $m_3 = 1, 2, \ldots, l_3$, is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/}$$

for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$.

These subprograms require that l_1, l_2, and l_3 be products of powers of 2, 3, and 5, that is, of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k = 1, 2, 3$, and where either $l_1 = 1$ or l_1 is even.

Usage

```
INTEGER*4       l1, l2, l3, ldx, mdx, iopt, ier
REAL*4          x(ldx, mdx, l3)
CALL SRC3FT(x, l1, l2, l3, ldx, mdx, iopt, ier)

INTEGER*4       l1, l2, l3, ldx, mdx, iopt, ier
REAL*8          x(ldx, mdx, l3)
CALL DRC3FT(x, l1, l2, l3, ldx, mdx, iopt, ier)
```

Input

x Array of data to be transformed.

For a forward real-to-complex transform, the real data point $z(n_1, n_2, n_3)$ is stored in $\mathbf{x}(n_1, n_2, n_3)$,
$n_1 = 1, 2, \ldots, \mathbf{l1}$, $n_2 = 1, 2, \ldots, \mathbf{l2}$,
$n_3 = 1, 2, \ldots, 2, \ldots, \mathbf{l3}$.

For an inverse complex-to-real transform, the real part of $Z(m_1,m_2,m_3)$ is stored in $\mathbf{x}(2\times m_1-1,m_2,m_3)$ and the imaginary part is stored in $\mathbf{x}(2\times m_1,m_2,m_3)$,
$m_1 = 1, 2, \ldots, \mathbf{l1}/2+1$, $m_2 = 1, 2, \ldots, \mathbf{l2}$,
$m_3 = 1, 2, \ldots, \mathbf{l3}$.

l1 Number of rows of data, of the form $\mathbf{l1} = 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either $\mathbf{l1} = 1$ or **l1** is even.

	l2	Number of columns of data, of the form **l2** = $2^{p_2}\ 3^{q_2}\ 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	l3	Number of planes of data, of the form **l3** = $2^{p_3}\ 3^{q_3}\ 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
	ldx	The leading dimension of array **x**, with **ldx** ≥ **l1**+2.
	mdx	The middle dimension of array **x**, with **mdx** ≥ **l2**.
	iopt	Option flag: **iopt** ≥ 0 Compute forward transform. **iopt** < 0 Compute inverse transform.
Output	**x**	The transformed data replaces the input if **ier** = 0 is returned. For a forward real-to-complex transform, the real part of $Z(m_1,m_2,m_3)$ is stored in **x**$(2\times m_1-1,m_2,m_3)$and the imaginary part is stored in **x**$(2\times m_1,m_2,m_3)$, m_1 = 1, 2, ..., **l1**/2+1, m_2 = 1, 2, ..., **l2**, m_3 = 1, 2, ..., **l3**. If needed, the remaining (**l1**/2 − 1) × **l2** × **l3** complex output values may be formed by using the conjugate-symmetry condition. For an inverse complex-to-real transform, the real output point $z(n_1,n_2,n_3)$ is stored in **x**(n_1,n_2,n_3), n_1 = 1, 2, ..., **l1**, n_2 = 1, 2, ..., **l2**, n_3 = 1, 2, ..., **l3**.
	ier	Status response: **ier** = 0 Normal return—transform successful. **ier** = −1 **l1** not of the required form. **ier** = −2 **l2** not of the required form. **ier** = −3 **l3** not of the required form. **ier** = −4 **ldx** < **l1**+2. **ier** = −5 **mdx** < **l2**. **ier** ≤ −6 Probable error in **ldx** or **mdx**.

Name CRCFTS/ZRCFTS
Simultaneous real-to-complex one-dimensional FFT

Purpose Given a number of one-dimensional data sets, these subprograms compute all of their one-dimensional forward real-to-complex or inverse complex-to-real discrete Fourier transforms using a radix 2-3-5 FFT algorithm optimized for real input or output, stored in a complex array. A pair of companion subprograms, SRCFTS and DRCFTS, performs the same operation, but in a space-conserving manner. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one real-to-complex or complex-to-real transform.

The one-dimensional complex forward DFT of a real data set $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, that is,

$$Im(Z(1)) = Im(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \overline{Z}(l/2+1-m), m = 1, 2, ..., l/2-1.$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the one-dimensional, real, scaled, inverse DFT of $Z(m)$ is defined by

$$z(n) = \frac{1}{l}\sum_{m=1}^{l} Z(m)e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms perform forward real-to-complex or inverse complex-to-real transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$, and where either $l = 1$ or l is even.

Usage

```
INTEGER*4     l, incl, n, incn, iopt, ier
COMPLEX*8     z(lenz)
CALL CRCFTS(z, l, incl, n, incn, iopt, ier)

INTEGER*4     l, incl, n, incn, iopt, ier
COMPLEX*16    z(lenz)
CALL ZRCFTS(z, l, incl, n, incn, iopt, ier)
```

Input

z Array containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **z** is a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **z** as a one-dimensional array, results in

$$\mathbf{lenz} = (\mathbf{l}-1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

The i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{z}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1).$$

For a forward real-to-complex transform, only real parts of **z** are used as input.

For an inverse complex-to-real transform, only the first **l**/2+1 complex data points of each data set are used as input.

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $q, r \ge 0$ and either **l** = 1 or **l** is even.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in array **z**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0	Compute forward transform.
iopt < 0	Compute inverse transform.

Output

z The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.

For a forward real-to-complex transform, each transformed data set satisfies the conjugate-symmetry condition.

For an inverse complex-to-real transform, the real result is stored in the real parts of **z** and the imaginary parts of z are set to zero.

ier Status response:

ier = 0	Normal return—transform successful.
ier = –1	**l** not of the required form.
ier = –2	**incl** ≤ 0.
ier = –3	**n** ≤ 0.
ier = –4	**incn** ≤ 0.
ier = –5	**l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **z** is a two- or three-dimensional array with each set of data being a one-dimensional section of the array, that is, all but one subscript is constant within a data set.

If **z** is a two-dimensional array of dimension **ldz** by **mdz**, and if the data sets are stored in the columns of **z**, then l ≤ ldz, **n** ≤ **mdz**, **incl** = 1, and **incn** = **ldz**. For example, use

CALL CRCFTS (z, l, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and the data sets are stored in the rows of **z**, then **l** ≤ **mdz**, **n** ≤ **ldz**, **incl** = ldz, and **incn** = 1. For example, use

CALL CRCFTS (z, l, ldz, n, 1, iopt, ier)

If **z** is a three-dimensional array of dimension **ldz** by **mdz** by **ndz**, then **incl** and **incn** will usually be 1, **ldz**, or ldz×**mdz**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use	**incl** = 1.
2nd subscript, use	**incl** = **ldz**.
3rd subscript, use	**incl** = **ldz**×**mdz**.

Similarly, if the subscript that varies between data sets is the

1st	subscript, use	**incn** = 1.
2nd	subscript, use	**incn** = **ldz**.
3rd	subscript, use	**incn** = **ldz**×**mdz**.

l, incl, n, and **incn** must be such that no two points of any data sets occupy the same element of **z**. These subprograms detect this situation and return **ier** = –5 if

$$\mathbf{incl} < \mathbf{n} \times \gcd(\mathbf{incl}, \mathbf{incn})$$

and

$$\mathbf{incn} < \mathbf{l} \times \gcd(\mathbf{incl}, \mathbf{incn}),$$

where gcd(.,.) is the greatest common divisor.

Example 1 Compute the forward discrete Fourier transform of 256 data sets data of length 1024, stored as columns of the COMPLEX*8 array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL CRCFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Example 2 Compute the inverse discrete Fourier transform of 1024 sets of conjugate-symmetric complex data of length 256, stored as the rows of COMPLEX*8 array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL CRCFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Name SRCFTS/DRCFTS
Simultaneous real-to-complex one-dimensional FFT

Purpose Given a number of one-dimensional real data sets, these subprograms compute nonredundant portions of all of their one-dimensional forward real-to-complex discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm optimized for real input. Alternatively, given the nonredundant parts of a number of conjugate-symmetric one-dimensional complex data sets, these subprograms compute the inverse complex-to-real discrete Fourier transform using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRCFTS and ZRCFTS, performs similar operations, but with the real or complex data presented in a complex array. These companion subprograms require more storage than the ones described here. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one real-to-complex or complex-to-real transform.

The one-dimensional forward discrete Fourier transform of a real data set $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, that is,

$$Im(Z(1)) = Im(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \bar{Z}(l/2+1-m), m = 1, 2, ..., l/2-1,$$

where $\bar{Z}$ is the complex conjugate of Z. Therefore, the nonredundant part consists of the first $l/2+1$ elements of Z. Only the nonredundant part of Z is computed or stored.

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the one-dimensional, real, scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$. Only the nonredundant part of Z is used.

These subprograms perform forward real-to-complex or inverse complex-to-real transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, that is, of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$, and where either $l = 1$ or l is even.

Usage

```
INTEGER*4       l, incl, n, incn, iopt, ier
REAL*4          x(lenx)
CALL SRCFTS(x, l, incl, n, incn, iopt, ier)

INTEGER*4       l, incl, n, incn, iopt, ier
REAL*8          x(lenx)
CALL DRCFTS(x, l, incl, n, incn, iopt, ier)
```

Input

x Array containing **n** one-dimensional data sets, each consisting of **l** real data points or the first **l**/2+1 complex data points of a conjugate-symmetric complex data set of length **l**, to be transformed. Typically, **x** is a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **x** as a one-dimensional array results in

$$\mathbf{lenx} = (\mathbf{l}+1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

For a forward real-to-complex transform, the i-th real data point of the j-th data set, $1 \leq i \leq \mathbf{l}$, $1 \leq j \leq \mathbf{n}$, is stored in

$$\mathbf{x}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1).$$

For an inverse complex-to-real transform, the real part of the i-th data point of the j-th data set, $1 \leq i \leq \mathbf{l}/2+1$, $1 \leq j \leq \mathbf{n}$, is stored in

$$\mathbf{x}((2\times i-2)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1)$$

and the imaginary part is stored in

$$\mathbf{x}((2\times i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1)$$

respectively.

	l	Number of data points in each complete data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $q, r \geq 0$ and either $\mathbf{l} = 1$ or **l** is even.
	incl	Storage increment between successive elements of the same data set, **incl** > 0. **incl** = 1 means that the data points of a real data set are stored contiguously in array **x**, or that the real and imaginary parts of the data points of a complex data set are stored alternatingly in contiguous elements of **x**.
	n	The number of data sets, **n** > 0.
	incn	Storage increment between corresponding data points of successive data sets, **incn** > 0.
	iopt	Option flag: **iopt** ≥ 0 Compute forward transform. **iopt** < 0 Compute inverse transform.
Output	**x**	The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0. For a forward real-to-complex transform, the real part of the i-th output point of the j-th data set, $1 \leq i \leq \mathbf{l}/2+1$, $1 \leq j \leq \mathbf{n}$, is stored in $\mathbf{x}((2 \times i - 2) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1)$ and the imaginary part is stored in $\mathbf{x}((2 \times i - 1) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1)$, respectively. If needed, the remaining $(\mathbf{l}/2 - 1) \times \mathbf{n}$ complex output values may be formed by using the conjugate-symmetry condition. For an inverse complex-to-real transform, the i-th real output point of the j-th data set, $1 \leq i \leq \mathbf{l}$, $1 \leq j \leq \mathbf{n}$, is stored in $\mathbf{x}((i - 1) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1)$.

ier	Status response:	
	ier = 0	Normal return—transform successful.
	ier = –1	**l** not of the required form.
	ier = –2	**incl** ≤ 0.
	ier = –3	**n** ≤ 0.
	ier = –4	**incn** ≤ 0.
	ier = –5	**l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **x** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, that is, all but one subscript will be constant within a data set.

If **x** is a two-dimensional array of dimension **ldx** by **mdx**, and if the data sets are stored in the columns of **x**, then **l+2** ≤ **ldx**, **n** ≤ **mdx**, **incl** = 1, and **incn** = **ldx**. For example,

CALL SRCFTS (x, l, 1, n, ldxy, iopt, ier)

If **x** is a two-dimensional array as above and the data sets are stored in the rows of **x**, then **l+2** ≤ **mdx**, **n** ≤ **ldx**, **incl** = **ldx**, and **incn** = 1. For example,

CALL SRCFTS (x, l, ldxy, n, 1, iopt, ier)

If **x** is a three-dimensional array of dimension **ldx** by **mdx** by **ndx**, then **incl** and **incn** will usually be 1, **ldx**, or **ldx**×**mdx**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st	subscript, use	**incl** = 1.
2nd	subscript, use	**incl** = **ldx**.
3rd	subscript, use	**incl** = **ldx**×**mdx**.

Similarly, if the subscript that varies between data sets is the

1st	subscript, use	**incn** = 1.
2nd	subscript, use	**incn** = **ldx**.
3rd	subscript, use	**incn** = **ldx**×**mdx**.

l, incl, n, and **incn** must be such that no two points of any data sets occupy the same elements of **x**. These subprograms detect this situation and return **ier** = –5 if

$$\mathbf{incl} < \mathbf{n} \times \gcd(\mathbf{incl}, \mathbf{incn})$$

and

$$\mathbf{incn} < (\mathbf{l} + 2) \times \gcd(\mathbf{incl}, \mathbf{incn}),$$

where gcd(.,.) is the greatest common divisor.

Example 1 Compute the forward real-to-complex discrete Fourier transform of 256 real data sets of length 1024. The real input data sets are stored as columns of REAL*4 array X whose dimensions are 1027 by 256. The complex output data sets are stored as columns of array X, with the real parts in rows 1, 3, 5, ..., 1025, and the imaginary parts in rows 2, 4, 6, ..., 1026.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
REAL*4    X(1027,256)
L = 1024
INCL = 1
N = 256
INCN = 1027
IOPT = 1
CALL SRCFTS (X,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Example 2 Compute the inverse complex-to-real discrete Fourier transform of 1024 sets of conjugate-symmetric complex data length 256. The real and imaginary parts of the first 129 complex data points of the input data sets are stored as the rows of array X whose dimensions are 1025 by 258, with the real parts in columns 1, 3, 5, ..., 257, and the imaginary parts in columns 2, 4, 6, ..., 258. The real output data sets will be stored by row in the first 256 columns of X.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
REAL*4    X(1025,258)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL SRCFTS (X,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

7 Correlation and Convolution Subprograms

Overview

This chapter explains how to use the VECLIB subprograms available for correlations and convolutions.

The following document provides supplemental material for this chapter:

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Chapter objectives

After reading this chapter you will know how to use the described subprograms to compute correlation and convolution.

What you need to know to use these subprograms

The subprograms presented here can be used to compute both discrete correlations and discrete convolutions. Refer to specific subprogram descriptions in "Subprograms for correlations and convolutions" on page 502 for details.

Subprograms for correlations and convolutions

The following sections describe subprograms included with VECLIB for correlations and convolutions.

Name

SCONV/DCONV
Correlation and convolution

Purpose

These subprograms compute the fully engaged portion of the discrete correlation or discrete convolution of two real data sequences. They can be used to compute the complete discrete correlation or convolution (the fully engaged portion plus the *tails*) by appending zeros to the ends of the longer of the operand vectors. Refer to "Example 2" on page 505.

If $x_i, i = 1, 2, ..., l$ and $w_j, j = 1, 2, ..., n$ are two data sequences, their discrete correlation y_k is defined by

$$y_k = \sum_i x_{k+i-1} w_i,$$

for $k = -n+1, -n+2, ..., l-1$, where the sum is taken over all indices i for which both x_{k+i-1} and w_i are defined.

If $x_i, i = 1, 2, ..., l$ and $w_j, j = 1, 2, ..., n$ are two data sequences, their discrete convolution z_k is defined by

$$z_k = \sum_i x_{k-i+1} w_i,$$

for $k = 1, 2, ..., l+n-1$, where the sum is taken over all indices i for which both x_{k-i+1} and w_i are defined.

These subprograms compute only the fully engaged portion of the correlation or convolution, that is, the part where the sums have exactly $min(l,n)$ terms. Hence, if $l \geq n$, they compute

$$\tilde{y}_k = \sum_{i=1}^{n} x_{k+i-1} w_i,$$

for $k = 1, 2, ..., m$, where $m = l - n + 1$. This is the correlation operation if w is stored or indexed in the same direction as x and is the convolution operation if x and w are stored or indexed in opposite directions.

Usage

```
INTEGER*4       incx, incw, incy, m, n
REAL*4          x(lenx), w(lenw), y(leny)
CALL SCONV(x, incx, w, incw, y, incy, m, n)

INTEGER*4       incx, incw, incy, m, n
REAL*8          x(lenx), w(lenw), y(leny)
CALL DCONV(x, incx, w, incw, y, incy, m, n)
```

Input

x Array containing the operand (or trace) vector x of length $m+n-1$. **lenx** = (**m**+**n**−2)×|**incx**|+1.

incx Increment for the array **x**, **incx** ≠ 0. x_i is stored in **x**((i−1)×**incx**+1). **incx** is normally positive whether computing the correlation or the convolution. Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(i). Refer to "Notes."

w Array containing the operator (or kernel) vector w of length n. **lenw** = (**n**−1)×|**incw**|+1. Refer to the description of **incw** for alternate usage of **w**.

incw Increment for the array **w**, **incw** ≠ 0. w_j is stored in w((j−1)×**incw**+1). **incw** is normally positive to compute the correlation and negative to compute the convolution.

Use **incw** = 1 if computing the correlation and vector w is stored contiguously in forward order in array **w**, that is, if w_j is stored in **w**(j). Refer to "Example 1" on page 504. Also, use **incw** = 1 if computing the convolution and vector w is stored contiguously in backward order in array **w**, that is, if w_j is stored in **w**(**n**+1−j).

Use **incw** = −1 if computing the convolution and vector w is stored contiguously in forward order in array **w**, that is, if w_j is stored in **w**(j). In this case, in order to index backward through the **w** array, the **w** argument is usually coded as **w**(**n**). Refer to "Notes" on page 504 and "Example 2" on page 505.

	incy	Increment for the array **y**, **incy** ≠ 0. $\tilde{y}_k$ is stored in **y**((k−1)×**incy**+1). **incy** will normally be positive whether computing the correlation or the convolution. Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if $\tilde{y}_k$ is stored in **y**(k). Refer to "Notes."
	m	The length of the y vector. **m** > 0.
	n	The length of the w vector. **n** > 0.
Output	**y**	The convolution or correlation vector y of length m. **leny** = (**m**−1)×|**incy**|+1.

Notes These subprograms do not use the BLAS indexing scheme described in "BLAS indexing conventions" on page 23. Indexing works the same way as the BLAS for a positive increment but, for a negative increment, these subprograms access data backward from the beginning of the array passed to it rather than from the end. To index backward through an array, as desired for a convolution, the ending location of the array should be passed to the subprogram. Refer to "Example 2" on page 505.

To compute the complete correlation or convolution vector, including both tails as well as the fully engaged portion, append n−1 zeros to each end of the x vector. The fully engaged portion of the correlation or convolution of the resulting vector is the complete correlation or convolution corresponding to the original x vector. Refer to"Example 2" on page 505

Example 1 Compute the fully engaged portion of the discrete correlation of the REAL*4 vectors x = (4, 1, 3, 5, 2) and w = (2, 1) stored in arrays X and W, respectively. In this instance, n = 2 and m+n−1 = 5, so m = 4.

```
INTEGER*4 INCX,INCW,INCY,M,N
REAL*4    X(5),W(2),Y(4)
DATA      X / 4.0 , 1.0 , 3.0 , 5.0, 2.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = 1
INCY = 1
CALL SCONV (X,INCX,W,INCW,Y,INCY,M,N)
```

The result is y = (9, 5, 11, 12).

Example 2 Compute the complete discrete convolution of the REAL*4 vectors $x = (1, 3, 5)$ and $w = (2, 1)$ stored in arrays X and W, respectively. To get the complete convolution, append $n-1 = 1$ zeros to each end of x, getting $\bar{x} = (0, 1, 3, 5, 0)$. Then, you have $n = 2$ and $m+n-1 = 5$, so $m = 4$.

```
INTEGER*4 INCX,INCW,INCY,M,N
REAL*4    X(5),W(2),Y(4)
DATA      X / 0.0 , 1.0 , 3.0 , 5.0, 0.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = -1
INCY = 1
CALL SCONV (X,INCX,W(N),INCW,Y,INCY,M,N)
```

The result is $y = (2, 7, 13, 5)$.

Name STCONV/DTCONV
Tapered correlation and convolution

Purpose These subprograms compute the discrete tapered correlation or discrete tapered convolution of two real data sequences.

If x_i and w_i, $i = 1, 2, \ldots, n$ are two data sequences, their discrete correlation y_k is defined by

$$y_k = \sum_i x_{k+i-1} w_i,$$

for $k = -n+2, -n+3, \ldots,$ n, where the sum is taken over all indices i for which both x_{k+i-1} and w_i are defined.

Again, if x_i and w_i, $i = 1, 2, \ldots, n$ are two data sequences, their discrete convolution z_k is defined by

$$z_k = \sum_i x_{k-i+1} w_i,$$

for $k = 1, 2, \ldots, 2n-1$, where the sum is taken over all indices i for which both x_{k-i+1} and w_i are defined.

These subprograms compute only m elements on one side of the correlation or convolution, specifically the part where $l \le k \le m$. Hence, they compute

$$\tilde{y}_k = \sum_{i=1}^{n-k+1} x_{k+i-1} w_i,$$

for $k = 1, 2, \ldots, m$. If $m > n$, $\tilde{y}_k = 0$ for $k = n+1, n+2, \ldots, m$. This is the tapered correlation operation if w is stored or indexed in the same direction as x, and is the tapered convolution operation if x and w are stored or indexed in opposite directions.

Usage

```
INTEGER*4       incx, incw, incy, m, n
REAL*4          x(lenx), w(lenw), y(leny)
CALL STCONV(x, incx, w, incw, y, incy, m, n)

INTEGER*4       incx, incw, incy, m, n
REAL*8          x(lenx), w(lenw), y(leny)
CALL DTCONV(x, incx, w, incw, y, incy, m, n)
```

Input

x Array containing the operand (or trace) vector x of length n. **lenx** = (**n**–1)×|**incx**|+1.

incx Increment for the array **x**, **incx** ≠ 0. x_i is stored in **x**((i–1)×**incx**+1). **incx** is normally positive whether computing the correlation or the convolution. Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(i). Refer to "Notes."

w Array containing the operator (or kernel) vector w of length n. **lenw** = (**n**–1)×|**incw**|+1. Refer to the description of **incw** for alternate usage of **w**.

incw Increment for the array **w**, **incw** ≠ 0. w_j is stored in **w**((j–1)×**incw**+1). **incw** is normally positive to compute the correlation and negative to compute the convolution.

Use **incw** = 1 if computing the correlation and vector w is stored contiguously in forward order in array **w**, that is, if w_j is stored in **w**(j). Refer to "Example 1" on page 508. Also, use **incw** = 1 if computing the convolution and vector w is stored contiguously in backward order in array **w**, that is, if w_j is stored in **w**(**n**+1–j).

Use **incw** = –1 if computing the convolution and vector w is stored contiguously in forward order in array **w**, that is, if w_j is stored in **w**(j). In this case, in order to index backward through the **w** array, the **w** argument is usually coded as **w**(**n**). Refer to "Notes" on page 508 and "Example 2" on page 508.

incy Increment for the array **y**, **incy** ≠ 0. y_k is stored in **y**((k–1)×**incy**+1). **incy** will normally be positive whether computing the correlation or the convolution. Use **incy** = 1 if the vector y is stored contiguously in array **y**, that is, if $\tilde{y}_k$ is stored in **y**(k). Refer to "Notes."

m The length of the y vector. **m** > 0.

n The length of the x and w vector. **n** > 0.

Output **y** The tapered convolution or correlation vector y of length m. **leny** = (**m**−1)×|**incy**|+1.

Notes These subprograms do not use the BLAS indexing scheme described in "BLAS Indexing Conventions" in the introduction to Chapter 2. Indexing works the same way as the BLAS for a positive increment, but for a negative increment, these subprograms access data backward from the beginning of the array passed to it rather than from the end. To index backward through an array, as desired for a convolution, the ending location of the array should be passed to the subprogram. Refer to "Example 2" on page 508.

Example 1 Compute the discrete tapered correlation of the REAL*4 vectors x = (4, 1) and w = (2, 1) stored in arrays X and W, respectively. In this instance, n = 2 and m = 4.

```
INTEGER*4 INCX,INCW,INCY,M,N
REAL*4    X(2),W(2),Y(4)
DATA      X / 4.0 , 1.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = 1
INCY = 1
CALL STCONV (X,INCX,W,INCW,Y,INCY,M,N)
```

The result is $\tilde{y}$ = (9, 2, 0, 0).

Example 2 Compute the discrete tapered convolution of the REAL*4 vectors x = (1, 3) and w = (2, 1) stored in arrays X and W, respectively.

```
INTEGER*4 INCX,INCW,INCY,M,N
REAL*4    X(2),W(2),Y(2)
DATA      X / 1.0 , 3.0 /
DATA      W / 2.0 , 1.0 /
M = 2
N = 2
INCX = 1
INCW = -1
INCY = 1
CALL STCONV (X,INCX,W(N),INCW,Y,INCY,M,N)
```

The result is $\tilde{y}$ = (7, 3).

8 Miscellaneous Routines

Overview

This chapter explains how to use VECLIB subprograms for a variety of operations that are not included in other chapters. It includes a description of subprograms that:

- Control shared-memory parallelism on systems with multiple processors
- Measure CPU or wall-clock time
- Generate random numbers
- Sort the elements of a vector into ascending or descending order
- Report errors detected in the usage of VECLIB subprograms

Chapter objective

After reading this chapter you will know how to use the subprograms described in "Miscellaneous subprograms" on page 510.

Associated documentation

The following documents provide supplemental material for this chapter:

Knuth, D.E. *The Art of Computer Programming*, Vol. 2: Seminumerical Algorithms. Menlo Park, California: Addison-Wesley. 1973.

Knuth, D.E. *The Art of Computer Programming*, Vol. 3: Sorting and Searching. Menlo Park, California: Addison-Wesley. 1973.

Miscellaneous subprograms

The following sections describe miscellaneous subprograms included with VECLIB.

Name

CPUTIME
Measure CPU time

Purpose

CPUTIME returns the total amount of CPU time used by a process. Both user time and system time are included. The time unit is seconds and the resolution of the timer is 0.01 seconds. CPUTIME also makes it easy to time a segment of code in a program. In a multithreaded execution, CPUTIME returns the sum of the CPU time from all the threads of the process, not the individual CPU time of a thread.

Usage

```
REAL*4  CPUTIME, time, tzero
time = CPUTIME(tzero)
```

Input

tzero	Starting time of a time interval you wish to measure. See Notes for details.
time	Elapsed CPU time in seconds minus **tzero**.

Notes

To measure the total CPU time used by a program, use **tzero** = 0.0.

To measure the amount of CPU time used by a segment of code in a program, use CPUTIME before and after the segment as follows:

```
REAL*4  CPUTIME, time, tzero
tzero = CPUTIME(0.0)
    segment of code to be timed
time = CPUTIME(tzero)
```

Name

F_BLASERROR
BLAS Standard error handler

Purpose

F_BLASERROR is an error handler for the BLAS Standard routines. It is called by a BLAS Standard routine when an input parameter has an invalid value. F_BLASERROR prints an error message and execution stops. You may supply a version of F_BLASERROR that modifies the action performed.

The following values of arguments are invalid for BLAS Standard routines and are flagged by the error handling routine:

- Any value of the operator arguments whose meaning is not specified in the language-dependent section of the BLAS Standard
- **incw** = 0 or **incx** = 0 or **incy** = 0 or **incz** = 0
- **lda, ldb, ldc,** or **ldt** < 1
- **lda** < *m* if the matrix is an *m-by-n* general matrix
- **lda** < *n* if the matrix is an *n-by-n* square, symmetric, or triangular matrix
- **lda** < **kl** + **ku** + 1, if the matrix is an *m-by-n* general band matrix
- **lda** < **k** + 1, if the matrix is an *n-by-n* symmetric or triangular band matrix with k superdiagonals or subdiagonals

Routine-specific error conditions are described in the respective subprogram documentation.

Usage

```
CHARACTER      SRNAME
CALL F_BLASERROR(SRNAME)
```

Input

SRNAME

The name of the routine that called F_BLASERROR.

Name MLIB_GETNUMTHREADS
Determine permitted extent of parallelism

Purpose This subprogram is used by all parallelized MLIB subprograms to determine the extent to which they may use shared-memory parallelism.

If companion subroutine MLIB_SETNUMTHREADS has not been called, or if it was last called with a zero argument, MLIB_GETNUMTHREADS uses the environment variable MLIB_NUMBER_OF_THREADS to determine the maximum number of threads that are allowed to execute parallelized MLIB subprograms called by the program. If MLIB_NUMBER_OF_THREADS is not set, only one thread is allowed; that is, MLIB subprograms are not run with parallelism.

If MLIB_SETNMTHREADS has been called and the argument for its last call was nonzero, MLIB_GETNUMTHREADS ignores MLIB_NUMBER_OF_THREADS and bases its return value on the last MLIB_SETNUMTHREADS call.

Usage

```
INTEGER*4       MLIB_GETNUMTHREADS, nthread
nthread = MLIB_GETNUMTHREADS()
```

Output **nthread**

The maximum number of threads that can be used at this time. **nthread** = 1 if any of the following apply:

- The machines has only one CPU
- The operating system or version of MLIB does not support thread-based parallelism
- The program is running on a single CPU
- The executable file is not marked for parallel execution
- The default maximum number of threads is one
- The program currently is multi-threaded

Name MLIB_SETNUMTHREADS
Control extent of parallelism

Purpose This subprogram is used to control the extent to which parallelized MLIB subprograms may use shared-memory parallelism.

nthread becomes internal state for MLIB_GETNUMTHREADS. Only one value of **nthread** is saved within a UNIX process; subsequent calls to MLIB_SETNUMTHREADS overwrite the previous value.

Usage

```
INTEGER*4       nthread
CALL MLIB_SETNUMTHREADS(nthread)
```

Input

nthread If **nthread** > 0, it specifies the maximum number of parallel threads that can be spawned in subsequent calls to parallelized MLIB subprograms.

If **nthread** = 0, MLIB_GETNUMTHREADS, on its next call, restores the default maximum number of threads from the MLIB_NUMBER_OF_THREADS environment variable, if defined.

If MLIB_NUMBER_OF_THREADS is not defined, the default maximum number of threads is set to one.

Name

RAN
VAX-compatible random numbers

Purpose

This subprogram produces uniform [0,1) pseudorandom numbers identical to those generated by the VAX random-number generator, RAN. RAN is identical in usage to its VAX counterpart. RAN returns one random number per call, while the companion subroutine RANV returns an array of random numbers per call. For details about RANV, refer to "RANV" on page 516.

These generators are based on the linear congruent method introduced by D. H. Lehmer in 1949; refer to *The Art of Computer Programming (1973)*. Given a starting seed, S_0, with $0 \le S_0 < 2^{32}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (69069 S_{n-1}) \bmod 2^{32}, n > 0.$$

This is generator 25 in Table 1 on page 102 of *The Art of Computer Programming (1973)*. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-32} S_n.$$

The period of these generators is 2^{32}, that is, they repeat the same sequence of pseudorandom numbers after about 4.3 billion numbers. Subprograms SRAN and DRAN, documented elsewhere in this chapter, are random number generators with a much longer period.

Usage

```
INTEGER*4       iseed
REAL*4          RAN, x
x = RAN(iseed)
```

Input

iseed	An initial seed to start a pseudorandom sequence, or the **iseed** returned by the previous call to RAN to continue a sequence.

Output

iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
x	The next pseudorandom number in the sequence.

Notes

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 32-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O11**, **SU,I10**, or **Z8**, respectively.

Example

Fill an array X that is 10 elements long with a sequence of pseudorandom numbers using the scalar subprogram RAN.

```
INTEGER*4 ISEED,N
REAL*4    RAN,X(10)
ISEED = 1234                ! STARTING SEED
N = 10
DO I = 1, N
    X(I) = RAN(ISEED)
END DO
```

This fills array X with the same sequence of pseudorandom numbers and also returns the same final value of ISEED, as the "Example" on page 517 for RANV.

Name RANV
VAX-compatible random numbers

Purpose This subprogram produces uniform [0,1) pseudorandom numbers identical to those generated by the VAX random-number generator, RAN. RANV returns an array of random numbers per call while the companion subroutine RAN returns one random number per call. For details about RAN, refer to "RAN" on page 514.

These generators are based on the linear congruent method introduced by D. H. Lehmer in 1949; refer to *The Art of Computer Programming (1973)*. Given a starting seed, S_0, with $0 \le S_0 < 2^{32}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (69069 S_{n-1} + 1) \bmod 2^{32}, n > 0$$

This is generator 25 in Table 1 on page 102 of *The Art of Computer Programming (1973)*. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-32} S_n$$

The period of these generators is 2^{32}, that is, they repeat the same sequence of pseudorandom numbers after about 4.3 billion numbers. Subprograms SRANV and DRANV, documented elsewhere in this chapter, are vectorized random number generators with a much longer period.

Usage

```
INTEGER*4       iseed, n
REAL*4          x(n)
CALL RANV(iseed, n, x)
```

Input

iseed	An initial seed to start a pseudorandom sequence, or the **iseed** returned by the previous call to RANV to continue a sequence.
n	The number of pseudorandom numbers to place in array **x**.

Output

iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
x	The next **n** pseudorandom numbers in the sequence are stored in the first **n** elements of **x**.

Notes

CALL RANV (iseed, n, x) produces the same **n** pseudorandom numbers as returned by **n** successive references to **RAN(iseed)**.

CALL RANV (iseed, n, x) produces the same value of **iseed** that would have resulted from the last of **n** successive references to **RAN(iseed)**.

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 32-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O11**, **SU,I10**, or **Z8**, respectively.

Example

Fill an array X that is 10 elements long with a sequence of pseudorandom numbers using the vector subprogram RANV.

```
INTEGER*4 ISEED,N
REAL*4    X(10)
ISEED = 1234                ! STARTING SEED
N = 10
CALL RANV (ISEED,N,X)
```

This fills array X with the same sequence of pseudorandom numbers and also returns the same final value of ISEED, as the "Example" on page 515 for RAN.

Name

SRAN/DRAN
Scalar long period random number generator

Purpose

These subprograms produce a sequence of uniformly-distributed [0,1) pseudorandom numbers with a period of 2^{48}. Functions SRAN and DRAN return one random number per call, while companion subprograms SRANV and DRANV, also documented in this chapter, return an array of random numbers per call.

These generators are based on the linear congruential method introduced by D. H. Lehmer in 1949; refer to *The Art of Computer Programming (1973)*. Given a starting seed, S_0, with $0 \le S_0 < 2^{48}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (31167285 S_{n-1} + 1) \bmod 2^{48}, n > 0.$$

This is generator 29 in Table 1 on page 102 of *The Art of Computer Programming (1973)*. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-48} S_n.$$

The period of these generators is 2^{48}; that is, they repeat the same sequence of pseudorandom numbers after about 281 trillion numbers.

Usage

```
INTEGER*8       iseed
REAL*4          SRAN, x
x = SRAN(iseed)

INTEGER*8       iseed
REAL*8          DRAN, x
x = DRAN(iseed)
```

Input

iseed	An initial seed to start a pseudorandom sequence, or the **iseed** returned by the previous call to the subprogram to continue a sequence.

Output

iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
x	The next pseudorandom number in the sequence.

Notes

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 48-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O16, SU, I15,** or **Z12,** respectively. Alternatively, in the absence of INTEGER*8, use 2O12, SU, 2I/O or 2Z8, respectively.

Example

Fill an array X that is 10 elements long with a sequence of pseudorandom numbers using the scalar subprogram SRAN.

```
INTEGER*4 N
INTEGER*8 ISEED
REAL*4    SRAN,X(10)
ISEED = 1234                ! STARTING SEED
N = 10
DO I = 1, N
    X(I) = SRAN(ISEED)
END DO
```

This fills array X with the same sequence of pseudorandom numbers and also returns the same final value of ISEED, as does the **"Example" on page 522 for SRANV.**

Name

SRANV/DRANV
Vector long period random number generator

Purpose

These subprograms produce a sequence of uniformly-distributed [0,1) pseudorandom numbers with a period of 2^{48}. Subroutines SRANV and DRANV return an array of random numbers per call, while companion functions SRAN and DRAN return one random number per call.

These generators are based on the linear congruent method introduced by D. H. Lehmer in 1949; refer to *The Art of Computer Programming (1973)*. Given a starting seed, S_0, with $0 \le S_0 < 2^{48}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (31167285 S_{n-1} + 1) \bmod 2^{48}, n > 0.$$

This is generator 29 in Table 1 on page 102 of *The Art of Computer Programming (1973)*. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-48} S_n.$$

The period of these generators is 2^{48}: that is, they repeat the same sequence of pseudorandom numbers after about 281 trillion numbers.

Usage

```
INTEGER*8       iseed
INTEGER*4       n, incx
REAL*4          x(lenx)
CALL SRANV(iseed, n, x, incx)

INTEGER*8       iseed
INTEGER*4       n, incx
REAL*8          x(lenx)
CALL DRANV(iseed, n, x, incx)
```

Input

iseed	An initial seed to start a pseudorandom sequence, or the **iseed** returned by the previous call to the subprogram to continue a sequence.
n	The number of pseudorandom numbers to place in array **x**.

incx Increment for the array **x**:

- **incx** ≥ 0: x is stored forward in array **x**; that is, x_i is stored in **x**((*i*−1)×**incx**+1).
- **incx** < 0: x is stored backward in array **x**; that is, x_i is stored in **x**((*i*−**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

iseed The seed that produces the next pseudorandom number in the sequence replaces the input seed.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector X. If **n** ≤ 0, then **x** is not referenced. Otherwise, the next **n** pseudorandom numbers in the sequence replaces the input.

Notes

CALL SRANV (iseed, n, x, incx) produces the same **n** pseudorandom numbers as returned by **n** successive references to **SRAN(iseed)**.

CALL SRANV (iseed, n, x, incx) produces the same value of **iseed** that would have resulted from the last of **n** successive references to **SRAN(iseed)**.

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 48-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O16**, **SU,I15**, or **Z12**, respectively. Alternatively, in the absence of INTEGER*8, use 2O12, SU, 2I/O or 2Z8, respectively

Example Fill an array X that is 10 elements long with a sequence of pseudorandom numbers using the vector subprogram SRANV.

```
INTEGER*8 ISEED
INTEGER*4 N,INCX
REAL*4    X(10)
ISEED = 1234                ! STARTING SEED
N = 10
INCX = 1
CALL SRANV (ISEED,N,X,INCX)
```

This fills array X with the same sequence of pseudorandom numbers and also returns the same final value of ISEED, as does the "Example" on page 519 for SRAN.

Name

SSORT/DSORT/ISORT
Sort array

Purpose

These subprograms use a quicksort algorithm to sort elements of a vector into ascending or descending algebraic order. The vector may be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

```
CHARACTER*(*) order
INTEGER*4     n, incx
REAL*4        x(lenx)
CALL SSORT(order, n, x, incx)
```

```
CHARACTER*(*) order
INTEGER*4     n, incx
REAL*8        x(lenx)
CALL DSORT(order, n, x, incx)
```

```
CHARACTER*(*) order
INTEGER*4     n, incx, x(lenx)
CALL ISORT(order, n, x, incx)
```

Input

order — Sort order option:

'A' or 'a'	Sort the elements of x into ascending algebraic order.
'D' or 'd'	Sort the elements of x into descending algebraic order.

n — Number of elements of vector x to be sorted. If $\mathbf{n} \leq 1$, the subprograms do not reference **x**.

x — Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the data to be sorted.

incx — Increment for the array **x**. x is stored forward in array **x** with increment $|\mathbf{incx}|$; that is, x_i is stored in $\mathbf{x}((i-1)\times|\mathbf{incx}|+1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**, that is, if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in Chapter 2.

Output

x — If $\mathbf{n} \leq 0$ or if **order** is not 'A', 'a', 'D', or 'd', then **x** is unchanged. Otherwise, the sorted result replaces the input.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **order** argument as *ascending* or *descending*. Refer to "Example 2" on page 524.

Example 1 Sort the elements of a REAL*4 vector x into ascending order, where x is a vector 100 elements long stored in a one-dimensional array X of dimension 200.

```
CHARACTER*(*) ORDER
INTEGER*4     N,INCX
REAL*4        X(200)
ORDER = 'A'
N = 100
INCX = 1
CALL SSORT (ORDER,N,X,INCX)
```

Example 2 Sort the elements of a REAL*8 vector x into descending order, where x is a vector 100 elements long stored in a one-dimensional array X of dimension 200.

```
CHARACTER*(*) ORDER
INTEGER*4     N,INCX
REAL*8        X(200)
CALL DSORT ('DESCENDING',100,X,1)
```

Name WALLTIME
Measure wall-clock time

Purpose WALLTIME returns the current wall-clock time. A base time is set during the first execution of WALLTIME, and the return value is the elapsed time from this base. The time unit is seconds, and the resolution of the timer is microseconds. WALLTIME also makes it easy to time a segment of code in a program.

Usage

```
REAL*4 WALLTIME, time, tzero
time = WALLTIME(tzero)
```

Input

tzero	Starting time of a time interval you wish to measure. See Notes for details.

Output

time	Elapsed wall-clock time in seconds minus **tzero**.

Notes To measure the wall-clock time used by a segment of code in a program, use WALLTIME before and after the segment as follows:

```
REAL*4 WALLTIME, time, tzero
tzero = WALLTIME(0.0)
    segment of code to be timed
time = WALLTIME(tzero)
```

Name

XERVEC
VECLIB error handler

Purpose

XERVEC is the error handler for many of the subprograms in the VECLIB library, as indicated in the "Notes" section in the subprogram descriptions. As supplied in VECLIB, XERVEC writes one of the following error messages onto the standard error file:

```
**********************************************************************
* XERVEC: subprogram name called with invalid value of argument number iarg *
**********************************************************************

**********************************************************************
* XERVEC: error detected by subprogram name: text of error message *
**********************************************************************

**********************************************************************
* XERVEC: error iarg detected by subprogram name: text of error message *
**********************************************************************
```

where *name* is the name of the subprogram in which the error was detected, *iarg* is the argument number of the offending argument, and *text of error message* is a character string. If the main program is in Fortran, a call traceback is also written onto the standard error file. XERVEC then terminates execution with a nonzero exit status.

You can supply a version of XERVEC that alters this action. All VECLIB subprograms that call XERVEC have a **RETURN** statement after the **CALL** statement, so your version could set a flag in a common block and **RETURN**. The flag could be tested in the program unit that calls the VECLIB subprogram.

Usage

```
CHARACTER*(*)  name, messag
INTEGER*4      iarg
CALL XERVEC(name, iarg, messag)
```

Input

name	The name of the subprogram in which the error was detected.
iarg	If *iarg* > 0, the error message is printed in the first form given above and *iarg* is the number of the argument that was found to be in error. If *iarg* = 0, the error message is printed in the second form shown and *iarg* is not part of the error message. If *iarg* < 0, the error message is printed in the third form shown and *iarg* is the error number.
messag	The text of the error message to be printed if **iarg** ≤ 0. Not used as input if **iarg** > 0.

Part 2

HP LAPACK

9 Introduction to LAPACK

Overview

LAPACK, a component of HP MLIB, is a collection of Fortran-callable subprograms that provides mathematical software for application programs involving linear algebra, including linear equations, least squares, eigenvalue problems, and the singular value decomposition. The package supersedes LINPACK and EISPACK. The National Science Foundation, the Defense Advanced Research Projects Agency, and the Department of Energy supported the development of the public-domain version of LAPACK, from which the Hewlett-Packard version of LAPACK was derived.

HP LAPACK fully conforms to the public domain version 3.0 of LAPACK in all user-visible usage conventions. The internal workings of some subprograms have been specialized for HP computers.

The information in Part 2 of this manual (*HP MLIB User's Guide*) supplements the *LAPACK Users' Guide* with information specific for HP LAPACK.

The topics covered in this manual are:

- Accessing LAPACK
- Optimizing
- Parallel processing
- Profiling LAPACK applications
- Rounding off
- Working storage
- Error handling
- Troubleshooting
- HP MLIB man pages

Although LAPACK was designed for use with Fortran programs, C programs can call LAPACK subprograms. Refer to Appendix A for more information.

Refer to "Associated documentation" on page 530 for information about the *LAPACK Users' Guide*.

Chapter objectives

After reading this chapter you will:

- Know how to access LAPACK library subprograms
- Understand how LAPACK works in a parallel computing environment
- Know how LAPACK interacts with the CXperf performance tool
- Understand LAPACK naming conventions
- Understand roundoff effects
- Understand how LAPACK handles errors
- Know how to access the online *HP MLIB* man pages
- Know what to do if you experience trouble using LAPACK subprograms

Associated documentation

The HP MLIB documentation set includes this manual and the *LAPACK Users' Guide*. The latter provides an introduction to the design of LAPACK as well as complete specifications for all the driver and computational routines.

The information in Part 2 of this manual (*HP MLIB User's Guide*, First Edition) supplements the *LAPACK Users' Guide* with information specific for HP LAPACK.

For the most recent edition of the *LAPACK Users' Guide*, refer to the Netlib repository at the following URL:

http://www.netlib.org/lapack/lug/lapack_lug.html

For a complete list of supplemental documentation, refer to the "Preface" of this manual.

Accessing LAPACK

The LAPACK library is available as an archive and a shared library. It consists of compiled subprograms ready for you to incorporate into your programs with the linker. To use LAPACK, include the appropriate declarations and CALL statements in your Fortran source program, and specify LAPACK as an object library at link time. Refer to "Accessing VECLIB" on page 4 for details about accessing the VECLIB library.

MLIB libraries are installed in the /opt/mlib/ directory. The entire path depends on your system type, as follows:

System Type	CPU Type	HP-UX Version	Address Width	Installation Directory
C-, J-, D-, K-,L-, R-, V- or N-Class	PA-8x00	11.0 or later	32-bit	/opt/mlib/lib/pa2.0
			64-bit	/opt/mlib/lib/pa20_64

The file name of the archive LAPACK library is liblapack.a. The LAPACK shared library is liblapack.sl.

Performance of your applications is better when you use archive libraries. However, if you need to keep executable files to a minimum size, you can use shared libraries on any pa2.0 system running the HP-UX 11.0 or later operating system.

NOTE The LAPACK library contains parallelized subprograms. Refer to Appendix C, "Parallelized Subprograms," for a list of HP LAPACK parallelized subprograms. If you run your program under HP-UX version 11.0 or greater with multiple processors, refer to "Parallel processing" on page 534 for additional information about linking your program.

Compiling and linking

There are several ways to link your program with the version of LAPACK tuned for the machine on which the program runs. By default, the **-llapack** option on the **f77, f90, cc**, or c89 command line that links your program selects the library that corresponds to 32-bit addressing on the machine on which you link.

f77 usage is similar to **f90** usage. However, f77 does not support 64-bit addressing. **cc** is the HP C compiler and c89 is the HP POSIX-conforming C compiler. The remainder of this book refers to the **cc** and **f90** compilers. **cc** or **f90** examples also apply to c89 and f70 compilers, respectively.

Method 1 below displays how to link a program that uses LAPACK on the same machine; methods 2 through 4 allow you to link a program for use on a specific machine type.

NOTE

When you use the **-aarchive_shared** flag on the compiler command line, it ensures that the compiler links with the archive library. If the archive library is not available, then it links the shared library. Using the archive library usually results in better performance of your application. However, if for example, you need to keep executable files to a minimum size, you can select the shared library first using the **-ashared_archive** flag. If you omit **-aarchive_shared** and **-ashared_archive**, the linker defaults to linking the shared library. Method 1 has an example using both flags.

1. To link a program that uses LAPACK for use on the same machine, use one of the following commands:

   f90 [*options*] *file* ... **-Wl,-aarchive_shared -llapack**

   cc [*options*] *file* ... **-Wl,-aarchive_shared -llapack -lcl**

   To select the shared library use the **-ashared_archive** flag as follows:

   f90 [*options*] *file* ... **-Wl,-ashared_archive -llapack**

   cc [*options*] *file* ... **-Wl,-ashared_archive -llapack -lcl**

2. Specify the entire path of the library file on the compiler command line that links your program. For example, to link your program with LAPACK for use with 32-bit addressing on a PA-8x00-based system, use one of the following:

   f90 [*options*] *file* ... **/opt/mlib/lib/pa2.0/liblapack.a**

   cc [*options*] *file* ... **/opt/mlib/lib/pa2.0/liblapack.a -lcl**

   Replace **liblapack.a** with liblapack.sl on your compiler command line if you want to link the shared library.

3. Use the **-llapack** option on the compiler command line that links your program, preceded by one of:

   -Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0

   -Wl,-aarchive_shared,-L/opt/mlib/lib/pa20_64

   For example, the command lines in Method 2 could be written:

   f90 [*options*] *file* ... **-Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0 -llapack**

   cc [*options*] *file* ... **-Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0 -llapack -lcl**

4. Set the LDOPTS environment variable to include one of:

   -aarchive_shared,-L/opt/mlib/lib/pa2.0

   -aarchive_shared,-L/opt/mlib/lib/pa20_64

   For example,

   setenv LDOPTS "-aarchive_shared,-L/opt/mlib/lib/pa2.0"

   Then use the **-llapack** option on the compiler command line that links your program:

   f90 [options] file ... **-llapack**

   cc [options] file ... **-llapack -lcl**

NOTE An LDOPTS specification takes precedence over the use of -Wl on the compiler command line. That is, if you use the **LDOPTS** environment variable to specify a library path, you cannot override that specification with a -Wl option on your compiler command line.

Linking both LAPACK and VECLIB libraries

If your program uses subprograms from both LAPACK and VECLIB, specify both **-llapack** and **-lveclib** on your command line. For example, on a PA-8x00-based machine with 32-bit addressing, link with

f90 [*options*] *file...* **-Wl,-aarchive_shared,-L/opt/mlib/lib/pa2.0 -llapack -lveclib**

Each of the LAPACK and VECLIB library files is complete in itself, meaning that you do not need to link one library because you used subprograms from another. This is because various subprograms are included in both libraries. For example, the VECLIB subroutine SGEMV is called by several LAPACK subprograms, and therefore is included in the LAPACK library. You may list libraries in any order on your link command line.

Optimizing

LAPACK has been optimized by using a highly efficient implementation of the Basic Linear Algebra Subprograms (BLAS), levels 1, 2 and 3, as well as a subset of the BLAS Standard. In addition, certain algorithmic improvements have been made, and several tunable parameters have been adjusted for good execution performance.

Parallel processing

Parallel processing is available on HP servers running the HP-UX version 11.0 or greater operating system. These systems can divide a single computational process into small streams of execution, called *threads*. The result is that you can have more than one processor executing on behalf of the same process. Also, refer to Appendix C, "Parallelized Subprograms," for a list of HP LAPACK parallelized subprograms.

You can enable or disable parallel processing at link time or at run time. A program does not use parallelism in LAPACK unless parallel processing is enabled both at link time and at runtime.

Linking for parallel or nonparallel processing

To enable parallel processing at link time, your link step must produce a multithreaded executable. You always get a multithreaded executable if you link with the Fortran 77, Fortran 90, or C compiler using the **+O3** and **+Oparallel** flags:

f90 [*options* including **+O3 +Oparallel**] *file...* **-Wl,-aarchive_shared -llapack**

cc [*options* including **+O3 +Oparallel**] *file...* **-Wl,-aarchive_shared -llapack -lcl**

To disable LAPACK's automatic parallelism at link time, omit the **+Oparallel** option:

f90 [*options*] *file* ... **-Wl,-aarchive_shared -llapack**

cc [*options*] *file* ... **-Wl,-aarchive_shared -llapack -lcl**

Controlling LAPACK parallelism at runtime

When you enable parallelism at link time, three methods are available at runtime to specify the extent of parallel processing to be allowed in MLIB.

- Use MLIB_NUMBER_OF_THREADS, a shell environment variable which allows you to enable parallelism within MLIB subprograms and to specify the maximum number of threads that may be used in parallel regions.

  Not setting MLIB_NUMBER_OF_THREADS has the same result as setting it to 1; that is, parallel processing is disabled within MLIB subroutines. Setting MLIB_NUMBER_OF_THREADS to the number of CPUs in the system, or greater, allows parallelized MLIB subprograms to use as many CPUs as are available to the process.

  The following command lines show the C shell syntax and Korn shell syntax to use when setting the variable to eight processors:

  For C shell:

  setenv MLIB_NUMBER_OF_THREADS 8

  For Korn shell:

  export MLIB_NUMBER_OF_THREADS=8

  MLIB_NUMBER_OF_THREADS is examined upon the first call to a parallelized LAPACK subprogram to establish the default parallel action within LAPACK.

  Use MLIB_SETNUMTHREADS to restore LAPACK parallel processing to its run-time default that was specified by MLIB_NUMBER_OF_THREADS. Refer to the mlib_setnumthreads(3m) man page for usage information.

- Use the subroutine MLIB_SETNUMTHREADS.

  You can call this subroutine at any time to set the maximum number of parallel threads used in subsequent VECLIB or LAPACK calls. The specified value overrides the absence of the MLIB_NUMBER_OF_THREADS environment variable or any value assigned to it.

- Use the environment variable MLIB_NUMBER_OF_THREADS at run time to control parallelism.

These controls set the maximum amount of parallelism that your program can use, and the LAPACK-specific mechanisms offer finer control within that maximum. Refer to "Performance benefits" on page 536 for more information.

Performance benefits

If LAPACK parallelism is enabled, each parallelized LAPACK subprogram determines at run time whether multiple processors are available. If so, it detects whether the program is already using multiple threads. It uses this information to choose automatically between a single- or parallel-processor algorithm.

If you are using an HP server with multiple processors, you can realize the performance benefits of parallel processing in three ways:

- Call any parallelized LAPACK subprogram. Let it use parallelism internally if it determines that it is appropriate to do so, based on such factors as problem size, system configuration, and user environment.
- Call LAPACK subprograms in a parallelized loop or region. To use this mechanism, you must be familiar with the techniques of parallel processing. Refer to the *Parallel Programming Guide for HP-UX Systems* for details.
- Use the MPI explicit parallel model. Refer to the *HP MPI User's Guide* and the MPI(1) man page for details.

LAPACK subprograms are reentrant, meaning that they can be called several times in parallel to do independent computations without one call interfering with another. You can use this feature to call LAPACK subprograms in a parallelized loop or region. The compiler does not automatically parallelize loops containing a function reference or subroutine call, but you can force it to do so by inserting compiler directives before the loop.

For example, the following Fortran code makes parallel calls to subprogram SGETRS:

```
C$DIR LOOP_PRIVATE (J)
C$DIR LOOP_PARALLEL
      DO 10 J=1, N
         CALL SGETRS ('N',N,1,A,LDA,IPIV,B(1,J), LDB,INFO(J))
   10 CONTINUE
```

While optimizing a parallel program, you might want to make parallel calls to an LAPACK subprogram to execute independent operations where the call statements are not in a loop. The Fortran compiler does not automatically parallelize code outside a loop, but you can use the BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives to instruct the compiler to parallelize such code.

If a parallelized LAPACK subprogram is called from a parallelized loop or region, internal parallelism is disabled.

Profiling LAPACK applications

CXperf is an interactive runtime performance analysis tool for programs compiled with HP ANSI C (cc), Fortran 90 (f90), and ANSI C++ (aCC) compilers. CXperf allows you to study the performance of a program for the purposes of optimizing, benchmarking, and debugging. To use CXperf, you must first compile your program with the **+pa** or **+pal** compiler option. These options instrument the compiled program to collect performance metrics for routines, loops, and compiler-generated parallel loops.

LAPACK is not instrumented for use with CXperf, so CXperf does not automatically collect or analyze performance information for LAPACK subroutines. However, you can use CXoi, the PA-RISC Object and Archive File Instrumenter, to insert CXperf instrumentation into the LAPACK libraries. When you use these instrumented libraries, the performance of LAPACK subprograms can be included in the analysis. See the cxoi(1) man page for more information.

CXperf is an optional product. For more information about CXperf, refer to the *CXperf Command Reference* and the *CXperf User's Guide* for details, or contact your HP sales representative.

Rounding off

LAPACK subprograms may use a different arithmetic order of evaluation than other implementations. Different roundoff characteristics can result. Accuracy of results is usually similar to other implementations, so using LAPACK should not materially affect the accumulation of roundoff errors in a complete application program. If it does, examine the mathematical analysis of the problem to determine if it is ill-conditioned. Ill-conditioned means that the small roundoff errors that are inadvertently introduced into any computation are magnified out of proportion to the desired result. Similarly, if results with and without LAPACK differ materially, both sets of answers are probably inaccurate and you should investigate further. If the program correctly applies stable computational algorithms, the problem itself is probably ill-posed.

Working storage

Many LAPACK subprograms require the user to supply one or more arrays to be used for work space. In some cases, the length of the required work space is not specified exactly in the subprogram documentation. For example, the work space length may be described as follows for a subroutine:

lwork	The length of array work. **lwork** ≥ max(1,**m**). For good performance, **lwork** must generally be larger. The optimum value of **lwork** for high performance is returned in work(1).

The performance of these subprograms can be improved, often dramatically, by providing at least the optimum amount of work space. This amount can be a complicated function of the problem size and job arguments, so you should make **lwork** large enough or compare it to the value returned in **work**(1).

Error handling

Most documented subprograms have a diagnostic argument **info**, which reports the success or failure of the computation as follows:

info = 0	Successful exit
info < 0	Invalid value of an argument—computation not completed
info > 0	Failure during the computation

All LAPACK driver and computational subprograms, and some auxiliary subprograms, check that input arguments such as **n** and **lda**, and option arguments such as **trans** and **uplo**, have valid numeric and character values respectively. If the k-th argument is invalid, the subprogram sets **info** = $-k$ and calls the error handling subprogram XERBLA.

The standard version of XERBLA writes an error message onto the standard error file, and terminates execution with a nonzero exit status. Thus, using the standard version of XERBLA, no LAPACK subprogram would ever return to the calling program with **info** < 0. You may supply a version of XERBLA that alters this action.

The description of each high-level subprogram defines the specific error code numbers and the related error conditions when **info** ≠ 0. Always check the output argument **info** after calling an LAPACK subprogram that has **info** as an argument, and take appropriate action if the output argument suggests a problem.

Refer to "XERBLA" on page 568 for details about the error handling subprogram.

Troubleshooting

Here are some suggestions to help you find the cause of a problem:

1. Verify that the subprogram usage in the program matches the subprogram specifications in the documentation. Pay attention to the number of arguments in the **CALL** statement and to the declarations of arrays and integer constants or variables that describe them. Write out all the arguments immediately before and after the **CALL** statement.
2. Make sure there really is a problem. For example, if an apparently incorrect answer is being computed, check to see if the answer does satisfy the problem as defined in the program. Also, for problems with more than one answer, LAPACK may produce a different answer or give the answers in a different order than expected. If the problem is ill-conditioned, LAPACK may not be able to compute a reliable answer. Error messages often suggest the cause of the problem.
3. Isolate the problem. If possible, write a small test program that encounters the same difficulty. For example, write the data causing the problem from the original program into the small one. In this way, you eliminate extraneous code from suspicion. If the problem area is large, try to pare it to a manageable size. For example, if a 50-by-50 linear system fails, try to produce a 2-by-2 system that fails in the same way.

HP MLIB man pages

HP MLIB man pages are installed in /opt/mlib/share/man. Set this path in your MANPATH environment variable to access man pages.

HP MLIB provides LAPACK man pages that contain:

- An introduction to LAPACK (lapack(3m))
- An introduction to each group of subprograms
- A reference page for each subprogram. HP MLIB is compliant with LAPACK 3.0—man pages for HP LAPACK subprograms are those provided with version 3.0 of the public domain LAPACK

For further explanation and a table of contents of reference entries for LAPACK, refer to the lapack(3) man page.

10 LAPACK Auxiliary Subprograms

Overview

This chapter describes selected LAPACK auxiliary subprograms. Although the auxiliary subprograms are a part of the public domain release of LAPACK, the Hewlett-Packard implementation of LAPACK does not support all of them. They are internal to the package and subject to change. The auxiliary subprograms described in this chapter, however, are supported as part of LAPACK and will exist in subsequent releases of the product. The operations covered are:

- Computing a norm of a matrix, for matrices stored in several different formats
- Reporting errors in a consistent manner

Chapter Objectives

After reading this chapter you will:

- Know what a vector norm and a matrix norm are and which matrix norms can be computed by LAPACK auxiliary subprograms
- Know how blocking parameters are set and used
- Know how to use the described subprograms

Associated documentation

The following documents provide supplemental material for this chapter:

- Anderson *et al.*, *LAPACK Users' Guide*. For the latest edition of the *LAPACK Users' Guide*, refer to the Netlib repository at the following URL:

  http://www.netlib.org/lapack/lug/lapack_lug.html

- Golub, G.H. and C.F. Van Loan. *Matrix Computations*, 2nd Edition. Baltimore, MD: The Johns Hopkins University Press. 1989.

What you need to know to use these subprograms

Norms of vectors and matrices

To use the norm-computing subprograms, you need to understand the basics of vector and matrix norms. For completeness, the following is a brief discussion of vector and matrix norms. Most standard texts on linear algebra cover the prerequisite material in greater detail.

Definition: A *vector norm* on $\mathbf{R}^n$, the vector space of n-dimensional real vectors, is a function $f:\mathbf{R}^n \rightarrow \mathbf{R}$ that has the following properties:

$f(x) \geq 0$	$x \in \mathbf{R}^n$
$f(x) = 0$	if and only if $x = 0$
$f(x+y) \leq f(x)+f(y)$	$x, y \in \mathbf{R}^n$
$f(\alpha x) = \lvert\alpha\rvert f(x)$	$\alpha \in \mathbf{R}, x \in \mathbf{R}^n$

Such a function is denoted with a double-bar notation: $f(x) = \|x\|$. Subscripts on the double bar are used to distinguish between various norms. The most important vector norms are the 1-, 2-, and ∞-norms, defined in Table 11-1.

The vector space of m-by-n matrices is isomorphic to the vector space of mn-dimensional vectors. Therefore, the definition of a matrix norm follows from the definition of a vector norm.

Definition: A *matrix norm* on $\mathbf{R}^{m\times n}$, the vector space of m-by-n real matrices, is a function $f:\mathbf{R}^{m\times n} \rightarrow \mathbf{R}$ that has the following properties:

$f(A) \geq 0$	$A \in \mathbf{R}^{m\times n}$
$f(A) = 0$	if and only if $A = 0$
$f(A+B) \leq f(A)+f(B)$	$A, B \in \mathbf{R}^{m\times n}$
$f(\alpha A) = \lvert\alpha\rvert f(A)$	$\alpha \in \mathbf{R}, A \in \mathbf{R}^{m\times n}$

As with vector norms, f is denoted with the double-bar notation: $f(A) = \|A\|$, again using subscripts to designate different matrix norms.

The formal definition of a matrix norm, given above, ignores the uses of matrices as operators in matrix-vector and matrix-matrix multiplication. Therefore, a matrix norm usually is required to satisfy several additional conditions related to such products.

Let $\|\ \|_\alpha$ be a vector norm on $\mathbf{R}^m$, $\|\ \|_\beta$ be a vector norm on $\mathbf{R}^n$, and $\|\ \|_{\alpha,\beta}$ be a matrix norm on $\mathbf{R}^{m\times n}$. The matrix norm is said to be *consistent with* the vector norm if

$$\|Ax\|_\beta \leq \|A\|_{\alpha,\beta}\|x\|_\alpha.$$

Let $\|\ \|_\alpha$ be a vector norm on $\mathbf{R}^m$, $\|\ \|_\beta$ be a vector norm on $\mathbf{R}^n$, and $\|\ \|_{\alpha,\beta}$ be a matrix norm on $\mathbf{R}^{m\times n}$. The matrix norm is said to be *induced by* or *subordinate to* the vector norm if

$$\|A\|_{\alpha,\beta} = \max_{x\neq 0} \frac{\|Ax\|_\beta}{\|x\|_\alpha}.$$

Finally, let $\|\ \|_\alpha$, $\|\ \|_\beta$, and $\|\ \|_\gamma$ be matrix norms on $\mathbf{R}^{m\times q}$, $\mathbf{R}^{m\times n}$, and $\mathbf{R}^{n\times q}$, respectively. The norms are *consistent* if the submultiplicative property

$$\|AB\|_\gamma \leq \|A\|_\alpha \|B\|_\beta$$

is satisfied for all $A \in \mathbf{R}^{m\times n}$ and $B \in \mathbf{R}^{n\times q}$.

The norm-computing auxiliary subprograms in LAPACK will evaluate the 1-, ∞-, Frobenius-, or Δ-norms of a matrix stored in a variety of forms, shown in Table 11-1.

Table 10-1 Norms of Vectors and Matrices

Name	Vector Norm	Matrix Norm
1-norm	$\|x\|_1 = \sum_i \|x_i\|$	$\|A\|_1 = max_j \sum_i \|a_{ij}\|$
2-norm	$\|x\|_2 = (\sum_i \|x_i\|^2)^{1/2}$	$\|A\|_2 = max_{x \neq 0} \|Ax\| / \|x\|$
∞-norm	$\|x\|_\infty = max_i \|x_i\|$	$\|A\|_\infty = max_i \sum_j \|a_{ij}\|$
Frobenius norm	$\|x\|_F = \|x\|_2$	$\|A\|_F = (\sum_{ij} \|a_{ij}\|^2)^{1/2}$
Δ-norm	—	$\|A\|_\Delta = max_{ij} \|a_{ij}\|$

The Frobenius matrix norm is not subordinate to the Frobenius vector norm. The Δ matrix norm is not subordinate to any vector norm, nor is it consistent with itself as a matrix norm.

Auxiliary subprograms

Following are the auxiliary subprograms included with LAPACK.

Name SLAMCH/DLAMCH
Return machine-dependent parameters

Purpose These subprograms return machine-dependent parameters, thus promoting machine independence in LAPACK.

Usage

```
CHARACTER*1     name
REAL*4          SLAMCH, value
value = SLAMCH(name)

CHARACTER*1     name
REAL*8          DLAMCH, value
value = DLAMCH(name)
```

Input

name Specifies which machine-dependent parameter is to be returned, as follows:

name = 'E' or 'e'	*eps*	The relative machine precision: the smallest number ε such that 1 + ε can be represented as a floating-point number with 1 + ε > 1.
name = 'S' or 's'	*sfmin*	The safe minimum: the smallest number such that *1/sfmin* does not overflow.
name = 'B' or 'b'	*base*	The base of the machine.
name = 'P' or 'p'	*prec*	*eps*×*base*.
name = 'N' or 'n'	*t*	The number of base-2 digits in the mantissa.
name = 'R' or 'r'	*rnd*	1.0 on machines where rounding occurs on addition; zero otherwise.
name = 'M' or 'm'	*emin*	The smallest exponent before underflow.
name = 'U' or 'u'	*rmin*	The underflow threshold: $base^{emin-1}$.
name = 'L' or 'l'	*emax*	The largest exponent before overflow.
name = 'O' or 'o'	*rmax*	The overflow threshold: $base^{emax}\times(1 - \varepsilon)$.

Output **value** The value of the requested machine-dependent parameter.

Notes Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the function reference may be improved by coding, for example, the **name** argument as 'Eps' for 'E' or 'Safemin' for 'S'.

Name

SLANGB/DLANGB/CLANGB/ZLANGB
Compute norm of general band matrix

Purpose

These subprograms compute the norm of an m-by-n general band matrix A. A band matrix is a matrix whose nonzero elements all lie near the principal diagonal. Specifically, $a_{ij} = 0$ if $i-j > kl$ or $j-i > ku$ for some integers kl and ku. The smallest such kl and ku for a given matrix are called the lower and upper bandwidths, respectively, and $k = kl+ku+1$ is the total bandwidth.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, you need only provide the elements within the band of A. Compared to storing the entire matrix, this can save memory if $kl+ku+1 < n$.

The following example illustrates the storage of general band matrices. Consider the following matrix A of order $n = 9$ and lower and upper bandwidths $kl = 2$ and $ku = 3$, respectively:

11	12	13	14	0	0	0	0	0
21	22	23	24	25	0	0	0	0
31	32	33	34	35	36	0	0	0
0	42	43	44	45	46	47	0	0
0	0	53	54	55	56	57	58	0
0	0	0	64	65	66	67	68	69
0	0	0	0	75	76	77	78	79
0	0	0	0	0	86	87	88	89
0	0	0	0	0	0	97	98	99

A is given in an array ab with at least $kl+ku+1 = 6$ rows and $n = 9$ columns as follows:

*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*

The asterisks in the ku-by-ku triangle at the upper left corner and in the kl-by-kl triangle at the lower right corner represent elements of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, then it is stored in **ab**($ku+1+i-j,j$). Therefore, the columns of A are stored in the columns of **ab**, the diagonals of A are stored in the rows of ab, and the principal diagonal is stored in row $ku+1$ of **ab**.

Note that this storage format omits the first *kl* rows reserved for fill-in in the general band storage for _GBSV and _GBTRF.

Usage

```
CHARACTER*1     norm
INTEGER*4       kl, ku, ldab, n
REAL*4          ab(ldab, n), work(n)
REAL*4          anorm, SLANGB
anorm = SLANGB( norm, n, kl, ku, ab, ldab, work)

CHARACTER*1     norm
INTEGER*4       kl, ku, ldab, n
REAL*8          ab(ldab, n), work(n)
REAL*8          anorm, DLANGB
anorm = DLANGB(norm, n, kl, ku, ab, ldab, work)

CHARACTER*1     norm
INTEGER*4       kl, ku, ldab, n
REAL*4          rwork(n)
COMPLEX*8       ab(ldab, n)
REAL*4          anorm, CLANGB
anorm = CLANGB(norm, n, kl, ku, ab, ldab, rwork)

CHARACTER*1     norm
INTEGER*4       kl, ku, ldab, n
REAL*8          rwork(n)
COMPLEX*16      ab(ldab, n)
REAL*8          anorm, ZLANGB
anorm = ZLANGB(norm, n, kl, ku, ab, ldab, rwork)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|\|A\|\|_F$ = the Frobenius norm.
norm = 'I' or 'i':	Compute $\|\|A\|\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|\|A\|\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

n The order of the matrix A. $n \geq 0$.

kl The number of subdiagonals within the band of A. $kl \geq 0$.

	ku	The number of superdiagonals within the band of A. **ku** ≥ 0.
	ab	The matrix A in band storage, in the first **kl+ku+1** rows. The j-th column of A is stored in the j-th column of array **ab** as follows: **ab**(**ku**+1+i–j,j) = $A(i,j)$ for max(1,j–**ku**) $\leq i \leq$ min(**n**,j+**kl**)
	ldab	The leading dimension of array **ab** in the calling program unit. **ldab** $\geq$ **kl+ku+1**.
Working Storage	**work, rwork**	Arrays used for work space. Not referenced unless **norm** = 'I' or '1'.
Output	**anorm**	The function value is the value of the requested norm of A.

Notes Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **norm** argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' or '1-norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANGE/DLANGE/CLANGE/ZLANGE
Compute norm of general matrix

Purpose

These subprograms compute a norm of a general *m*-by-*n* matrix *A*.

Usage

```
CHARACTER*1     norm
INTEGER*4       lda, m, n
REAL*4          a(lda, n), work(n)
REAL*4          anorm, SLANGE
anorm = SLANGE(norm, m, n, a, lda, work)

CHARACTER*1     norm
INTEGER*4       lda, m, n
REAL*8          a(lda, n), work(n)
REAL*8          anorm, DLANGE
anorm = DLANGE(norm, m, n, a, lda, work)

CHARACTER*1     norm
INTEGER*4       lda, m, n
REAL*4          rwork(n)
COMPLEX*8       a(lda, n)
REAL*4          anorm, CLANGE
anorm = CLANGE(norm, m, n, a, lda, rwork)

CHARACTER*1     norm
INTEGER*4       lda, m, n
REAL*8          rwork(n)
COMPLEX*16      a(lda, n)
REAL*8          anorm, ZLANGE
anorm = ZLANGE(norm, m, n, a, lda, rwork)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|A\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|A\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|A\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

m The number of rows of the matrix *A*. **n** ≥ 0.

n The number of columns of the matrix *A*. **n** ≥ 0.

	a	The m-by-n matrix A.
	lda	The leading dimension of array **a** in the calling program unit. **lda** ≥ **max(1,m)**.
Working Storage	**work, rwork**	Arrays used for work space. Not referenced unless **norm** = 'I' or '1'.
Output	**anorm**	The function value is the value of the requested norm of *A*.

Notes Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **norm** argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' or '1-norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANGT/DLANGT/CLANGT/ZLANGT
Compute norm of general tridiagonal matrix

Purpose

These subprograms compute a norm of a general tridiagonal matrix A. A matrix $A = (a_{ij})$ is tridiagonal if its nonzero elements lie only on the principal diagonal ($i = j$), the subdiagonal ($i = j+1$), and the superdiagonal ($i = j-1$) of the matrix.

Matrix Storage

The following example illustrates the storage of general tridiagonal matrices. Consider the following tridiagonal matrix of order $n = 7$:

$$\begin{bmatrix} 11 & 12 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 45 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 67 \\ 0 & 0 & 0 & 0 & 0 & 76 & 77 \end{bmatrix}$$

The subdiagonal is stored in array **dl**, the principal diagonal is stored in array **d**, and the superdiagonal is stored in array **du**, as follows:

i	**dl**(i)	**d**(i)	**du**(i)
1	21	11	12
2	32	22	23
3	43	33	34
4	54	44	45
5	65	55	56
6	76	66	67
7		77	

Usage

```
CHARACTER*1   norm
INTEGER*4     n
REAL*4        d(n), dl(n-1), du(n-1)
REAL*4        anorm, SLANGT
anorm = SLANGT(norm, n, dl, d, du)

CHARACTER*1   norm
INTEGER*4     n
REAL*8        d(n), dl(n-1), du(n-1)
REAL*8        anorm, DLANGT
anorm = DLANGT(norm, n, dl, d, du)
```

```
CHARACTER*1     norm
INTEGER*4       n
COMPLEX*8       d(n), dl(n-1), du(n-1)
REAL*4          anorm, CLANGT
anorm = CLANGT(norm, n, dl, d, du)

CHARACTER*1     norm
INTEGER*4       n
COMPLEX*16      d(n), dl(n-1), du(n-1)
REAL*8          anorm, ZLANGT
anorm = ZLANGT(norm, n, dl, d, du)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|A\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|A\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|A\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

n The order of the matrix A. $\mathbf{n} \geq 0$.

dl The $n-1$ subdiagonal elements of A.

d The diagonal elements of A.

du The $n-1$ superdiagonal elements of A.

Output

anorm The function value is the value of the requested norm of A.

Notes

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the CALL statement may be improved by coding the norm argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-Norm' or 'Norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANSB/DLANSB/CLANHB/CLANSB/ZLANHB/ZLANSB
Compute norm of symmetric or Hermitian band matrix

Purpose

These subprograms compute a norm of a real or complex symmetric or complex Hermitian band matrix. A real matrix is symmetric if $A = A^T$, its transpose; a complex matrix is Hermitian if $A = A^*$, its conjugate transpose.

Tridiagonal matrices are the special case $kd = 1$. They can be handled more efficiently by the LAPACK subprograms SLANST, DLANST, CLANHT, and ZLANHT.

The structure of A is indicated by the name of the subprogram used:

SLANSB	or	DLANSB	A is a real symmetric matrix.
CLANSB	or	ZLANSB	A is a complex symmetric matrix.
CLANHB	or	ZLANHB	A is a complex Hermitian matrix.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and since either triangle of A may be obtained from the other, you need only provide the band within one triangle of A. Compared to storing the entire matrix, this can save memory in two ways: Only the elements within the band are stored and, of them, only the upper or the lower triangle.

The following examples illustrate the storage of symmetric or Hermitian band matrices. Consider the following matrix A of order $n = 7$ and half bandwidth $kd = 2$:

$$\begin{bmatrix} 11 & 12 & 13 & 0 & 0 & 0 & 0 \\ 12 & 22 & 23 & 24 & 0 & 0 & 0 \\ 13 & 23 & 33 & 34 & 35 & 0 & 0 \\ 0 & 24 & 34 & 44 & 45 & 46 & 0 \\ 0 & 0 & 35 & 45 & 55 & 56 & 57 \\ 0 & 0 & 0 & 46 & 56 & 66 & 67 \\ 0 & 0 & 0 & 0 & 57 & 67 & 77 \end{bmatrix}$$

Upper triangular storage

The upper triangle of A is stored in an array **ab** with at least $kd+1 = 3$ rows and 7 columns as follows:

$$\begin{bmatrix} * & * & 13 & 24 & 35 & 46 & 57 \\ * & 12 & 23 & 34 & 45 & 56 & 67 \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 \end{bmatrix}$$

The asterisks represent elements in the *kd*-by-*kd* triangle at the upper left corner of **ab** that are not referenced. If a_{ij} is an element within the band of the upper triangle of A, it is stored in ab(kd+1+i–j,j). Therefore, the columns of the upper triangle of A are stored in the columns of **ab**, and the diagonals of the upper triangle of A are stored in the rows of **ab**.

Lower triangular storage

The lower triangle of A is stored in the array **ab** as follows:

11	22	33	44	55	66	77
12	23	34	45	56	67	*
13	24	35	46	57	*	*

The asterisks represent elements in the *kd*-by-*kd* triangle at the lower right corner of **ab** that are not referenced. If a_{ij} is an element within the band of the lower triangle of A, it is stored in ab(kd+1+i–j,j). Therefore, the columns of the lower triangle of A are stored in the columns of **ab**, and the diagonals of the lower triangle of A are stored in the rows of **ab**.

Usage

```
CHARACTER*1   norm, uplo
INTEGER*4     kd, ldab, work
REAL*4        ab(ldab, *), work(*)
REAL*4        anorm, SLANSB
anorm = SLANSB(norm, uplo, kd, ab, ldab, work)

CHARACTER*1   norm, uplo
INTEGER*4     kd, ldab, work
REAL*8        ab(ldab, *), work(*)
REAL*8        anorm, DLANSB
anorm = DLANSB(norm, uplo, kd, ab, ldab, work)

CHARACTER*1   norm, uplo
INTEGER*4     kd, ldab, work
REAL*4        rwork(*)
COMPLEX*8     ab(ldab, *)
REAL*4        anorm, CLANHB
anorm = CLANHB(norm, uplo, kd, ab, ldab, rwork)
```

```
CHARACTER*1       norm, uplo
INTEGER*4         kd, ldab, n
REAL*4            work(n)
COMPLEX*8         ab(ldab, n)
REAL*4            CLANSB, rm
anorm = CLANSB(norm, uplo, n, kd, ab, ldab, rwork)

CHARACTER*1       norm, uplo
INTEGER*4         kd, ldab, n
REAL*8            work(n)
COMPLEX*16        ab(ldab, n)
REAL*8            ZLANHB, rm
anorm = ZLANHB(norm, uplo, n, kd, ab, ldab, rwork)

CHARACTER*1       norm, uplo
INTEGER*4         kd, ldab, n
REAL*8            work(n)
COMPLEX*16        ab(ldab, n)
REAL*8            ZLANSB, rm
anorm = ZLANSB(norm, uplo, n, kd, ab, ldab, rwork)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|A\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|A\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|A\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

uplo Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix A is stored, as follows:

uplo = 'U' or 'u'	The upper triangular part of A is stored.
uplo = 'L' or 'l'	The lower triangular part of A is stored.

n The order of the matrix A. $\mathbf{n} \geq 0$.

kd The number of super-diagonals of the matrix A if uplo = 'U' or 'u', or the number of sub-diagonals if uplo = 'L' or 'l'. $\mathbf{kd} \geq 0$.

ab — The upper or lower triangle of the symmetric or Hermitian band matrix A, stored in the first **kd**+1 rows of the array. The j-th column of A is stored in the j-th column of array **ab** as follows:

If **uplo** = 'U' or 'u', ab(**kd**+1+i–j,j) = $A(i,j)$ for max(1,j–kd) ≤ i ≤ j;

If **uplo** = 'L' or 'l', ab(1+i–j,j) = $A(i,j)$ for j ≤ i ≤ min(**n**,j+kd).

ldab — The leading dimension of array **ab** in the calling program unit. **ldab** ≥ **kd+1**.

Working Storage

work, rwork — Arrays used for work space. Not referenced unless **norm** = '1' or 'I' or 'i' or 'O' or 'o'.

Output

anorm — The function value is the value of the requested norm of A.

Notes

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **norm** argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' or '1-norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANSP/DLANSP/CLA... P/CLANSP/.../ZLANSP
Compute norm of symm... c or Hermitian packed matrix

Purpose

These subprograms com... e a norm of a real or complex symmetric or complex Hermitian matrix A t... stored in an array in packed form.

A matrix is symmetric if ... = A^T, its transpose; a matrix is Hermitian if $A = A^*$, its conjugate tr... pose.

The structure of A is in... ed by the name of the subprogram used:

SLANSP	or	... NSP	A is a real symmetric packed matrix.
CLANSP	or	Z... NSP	A is a complex symmetric packed matrix.
CLANHP	or	Z... NHP	A is a complex Hermitian packed matrix.

Matrix Storage

Because either triangle o... A may be obtained from the other, you need only provide one triangle of A... ither the upper or the lower triangle. Compared to storing the entire matrix ... u save memory by supplying that triangle stored column-by-column in pac... l form in a 1-dimensional array.

The following examples ... strate the packed storage of symmetric or Hermitian matrices.

Upper triangular storage

If the upper triangle of A

$$\begin{matrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 24 \\ & & 33 & 34 \\ & & & 44 \end{matrix}$$

then A is packed colum... -column into an array **ap** as follows:

k	1	...	3	4	5	6	7	8	9	10
ap(k)	11	...	22	13	23	33	14	24	34	44

Upper triangular matrix ... ement a_{ij} is stored in array element **ap**(i+(j×(j–1))/2).

Lower triangular storage

If the lower triangle of A is

$$\begin{matrix} 11 & & & \\ 21 & 22 & & \\ 31 & 32 & 33 & \\ 41 & 42 & 43 & \mathbf{44} \end{matrix}$$

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower triangular matrix element a_{ij} is stored in array element **ap**(i+((j–1)×(2n–j))/2).

Usage

```
CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*4          ap((n*(n+1))/2), work(n)
REAL*4          anorm, SLANSP
anorm = SLANSP(norm, uplo, n, ap, work)

CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*8          ap((n*(n+1))/2), work(n)
REAL*8          anorm, DLANSP
anorm = DLANSP(norm, uplo, n, ap, work)

CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*4          rwork(n)
COMPLEX*8       ap((n*(n+1))/2)
REAL*4          anorm, CLANHP
anorm = CLANHP(norm, uplo, n, ap, rwork)

CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*4          rwork(n)
COMPLEX*16      ap((n*(n+1))/2)
REAL*4          anorm, CLANSP
anorm = CLANSP(norm, uplo, n, ap, rwork)
```

```
CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*8          rwork(n)
COMPLEX*16      ap(n*(n+1))/2)
REAL*8          anorm, ZLANHP
anorm = ZLANHP(norm, uplo, n, ap, rwork)

CHARACTER*1     norm, uplo
INTEGER*4       n
REAL*8          rwork(n)
COMPLEX*16      ap(n*(n+1))/2)
REAL*8          anorm, CLANSP
anorm = ZLANSP(norm, uplo, n, ap, rwork)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|A\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|A\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|A\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

uplo Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix A is stored, as follows:

uplo = 'U' or 'u'	The upper triangular part of A is stored.
uplo = 'L' or 'l'	The lower triangular part of A is stored.

n The order of the matrix A. $\mathbf{n} \geq 0$.

ap The upper or lower triangular part of the symmetric or Hermitian matrix A, packed columnwise in a linear array as follows:

If uplo = 'U' or 'u', $\mathbf{ap}(i + ((j-1) \times j)/2) = A(i,j)$ for $1 \leq i \leq j$;

If uplo = 'L' or 'l', $\mathbf{ap}(i + ((j-1) \times (2\mathbf{n}-j))/2) = A(i,j)$ for $j \leq i \leq n$.

Working Storage	**work, rwork**	Arrays u[illegible] for work space. Not referenced unless **norm** = '[illegible]' 'I' or 'i' or 'O' or 'o'.
Output	**anorm**	The funct[illegible] value is the value of the requested norm of A.

Notes Actual character arguments in a s[illegible]routine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by co[illegible]g the **norm** argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' [illegible] '1-norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANST/DLANST/CLANHT/ZLANHT
Compute norm of symmetric or Hermitian tridiagonal matrix

Purpose

These subprograms compute a norm of a real symmetric or complex Hermitian tridiagonal matrix A. A real matrix is symmetric if $A = A^T$, its transpose; a complex matrix is Hermitian if $A = A^*$, its conjugate transpose.

A matrix $A = (a_{ij})$ is tridiagonal if its nonzero elements lie only on the principal diagonal ($i = j$), the subdiagonal ($i = j+1$), and the superdiagonal ($i = j-1$) of the matrix.

Matrix Storage

The following example illustrates the storage of symmetric or Hermitian tridiagonal matrices. Consider the following symmetric tridiagonal matrix of order $n = 7$:

$$\begin{bmatrix} 11 & 21 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 32 & 0 & 0 & 0 & 0 \\ 0 & 32 & 33 & 43 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 54 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 65 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 76 \\ 0 & 0 & 0 & 0 & 0 & 76 & 77 \end{bmatrix}$$

The subdiagonal is stored in array **e** and the principal diagonal is stored in array **d**, as follows:

i	**e**(i)	**d**(i)
1	21	11
2	32	22
3	43	33
4	54	44
5	65	55
6	76	66
7		77

Usage

```
CHARACTER*1     norm
INTEGER*4       n
REAL*4          d(n), e(n-1)
REAL*4          anorm, SLANST
anorm = SLANST(norm, n, d, e)

CHARACTER*1     norm
INTEGER*4       n
REAL*8          d(n), e(n-1)
REAL*8          anorm, DLANST
anorm = DLANST(norm, n, d, e)

CHARACTER*1     norm
INTEGER*4       n
REAL*4          d(n)
COMPLEX*8       e(n-1)
REAL*4          anorm, CLANHT
anorm = CLANHT(norm, n, d, e)

CHARACTER*1     norm
INTEGER*4       n
REAL*8          d(n)
COMPLEX*16      e(n-1)
REAL*8          anorm, ZLANHT
anorm = ZLANHT(norm, n, d, e)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'M', 'm', 'E', or 'e'	Compute $\|\|A\|\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|\|A\|\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|\|A\|\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

n The order of the matrix A. $\mathbf{n} \geq 0$.

d The diagonal elements of the tridiagonal matrix A.

e The n-1 subdiagonal elements of the tridiagonal matrix A.

Output **anorm** The function value is the value of the requested norm of A.

Notes Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the CALL statement may be improved by coding the norm argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' or '1-norm' for 'O', or 'Max-Element' for 'M'.

Name

SLANSY/DLANSY/CLANHE/CLANSY/ZLANHE//ZLANSY
Compute norm of symmetric or Hermitian matrix

Purpose

These subprograms compute a norm of a real or complex symmetric or complex Hermitian matrix A.

A matrix is symmetric if $A = A^T$, its transpose; a matrix is Hermitian if $A = A^*$, its conjugate transpose.

The structure of A is indicated by the name of the subprogram used:

SLANSY	or	DLANSY	A is a real symmetric matrix.
CLANSY	or	ZLANSY	A is a complex symmetric matrix.
CLANHE	or	ZLANHE	A is a complex Hermitian matrix.

Matrix Storage

Because either triangle of A may be obtained from the other, you need only provide one triangle of A. You can supply either the upper or the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The other triangle of the array is not referenced.

Usage

```
CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*4          a(lda, *), work(n)
REAL*4          anorm, SLANSY
anorm = SLANSY(norm, uplo, n, a, lda, work)

CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*8          a(lda, *), work(n)
REAL*8          anorm, DLANSY
anorm = DLANSY(norm, uplo, n, a, lda, work)

CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*4          rwork(*)
COMPLEX*8       a(lda, *)
REAL*4          anorm, CLANHE
anorm = CLANHE(norm, uplo, n, a, lda, rwork)

CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*4          rwork(*)
COMPLEX*8       a(lda, *)
REAL*4          anorm, CLANSY
anorm = CLANSY(norm, uplo, n, a, lda, rwork)
```

```
CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*8          rwork(n)
COMPLEX*16      a(lda, n)
REAL*8          anorm, ZLANHE
anorm = ZLANHE(norm, uplo, n, a, lda, rwork)
```

```
CHARACTER*1     norm, uplo
INTEGER*4       lda, n
REAL*8          rwork(n)
COMPLEX*16      a(lda, n)
REAL*8          anorm, ZLANSY
anorm = ZLANSY(norm, uplo, n, a, lda, rwork)
```

Input

norm Specifies which norm is to be computed, as follows:

norm = 'F', 'f', 'E', or 'e'	Compute $\|\|A\|\|_F$ = the Frobenius norm.
norm = 'I' or 'i'	Compute $\|\|A\|\|_\infty$ = maximum row sum.
norm = '1', 'O', or 'o'	Compute $\|\|A\|\|_1$ = maximum column sum.
norm = 'M' or 'm'	Compute $max(\|A_{ij}\|)$.

uplo Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix A is stored, as follows:

uplo = 'U' or 'u'	The upper triangular part of A is stored.
uplo = 'L' or 'l'	The lower triangular part of A is stored.

n The order of the matrix A. **n** ≥ 0.

a The symmetric or Hermitian matrix A, as follows:

If **uplo** = 'U' or 'u', the leading **n**-by-**n** upper triangular part of **a** contains the upper triangular part of the matrix A, and the strictly lower triangular part of **a** is not referenced.

If **uplo** = 'L' or 'l', the leading **n**-by-**n** lower triangular part of **a** contains the lower triangular part of the matrix A, and the strictly upper triangular part of **a** is not referenced.

	lda	The leading dimension of array **a** in the calling program. **lda** ≥ max(1,**n**).
Working Storage	**work**, **rwork**	Arrays used for work space. Not referenced unless **norm** = 'I' or 'i' or 'O' or 'o'.
Output	**anorm**	The function value is the value of the requested norm of *A*.

Notes Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **norm** argument as 'Frobenius' for 'F', 'Infinity-Norm' for 'I', 'One-norm' for 'O', or 'Max-Element' for 'M'.

Name XERBLA
Error handler

Purpose This subprogram is the error handler for many LAPACK subprograms, as indicated in the "Notes" section in the applicable subprogram descriptions. As supplied in LAPACK, XERBLA writes the following error message onto the standard error file:

```
*****************************************************************************
* XERBLA: [illegible]program [illegible]ed with invalid value of argument number iarg *
*****************************************************************************
```

where *name* is the name of the subprogram in which the error was detected, and *iarg* is the argument number of the offending argument. For example, in SGBSV, **n** is argument number 1 and **kl** is argument number 2. If the main program is in Fortran and is executed under SPP-UX, a call traceback is also written onto the standard error file. XERBLA does not write a call traceback when used on HP-UX systems. XERBLA then terminates execution with a nonzero exit status.

You can supply a version of XERBLA that alters this action. All LAPACK subprograms that call XERBLA have a **RETURN** statement after the **CALL XERBLA** statement, so if your version of XERBLA exits with a **RETURN** statement, you could detect the error by examining the **info** flag after each LAPACK subprogram call.

Other subprograms, such as the Level 2 and 3 BLAS, also call XERBLA. All BLAS and VECLIB subprograms that call XERBLA also follow the **CALL XERBLA** statement with a **RETURN** statement. However, many of those subprograms do not have a status response variable such as **info** in their argument list to alert the caller. If you write an XERBLA that does not end with a **STOP** statement, you need some other mechanism to detect errors occurring in non-LAPACK subprograms. One such mechanism is a flag in a common block that is set by your XERBLA and tested by the calling program after calls where errors could be detected.

Usage

```
CHARACTER*6    name
INTEGER*4      iarg
CALL XERBLA(name, iarg)
```

Input

name The name of the subprogram in which the error was detected.

iarg The number of the argument that was found to be in error.

Notes This subprogram conforms to specifications of the Level 2 and 3 BLAS and LAPACK.

A Calling MLIB routines from C

Overview

Subprogram calls in Fortran use the same calling convention as used by other Hewlett-Packard language processors. This HP standard permits MLIB subprograms, that are written in Fortran or assembly language compatible with Fortran, to be called from programs written in C. This appendix describes the calling conventions necessary to call MLIB subprograms—VECLIB and LAPACK—from C programs.

The topics covered include:

- General interlanguage programming rules
- Header files
- Examples
- Linking VECLIB subprograms with your program
- Error handling

General interlanguage programming rules

The following are rules for calling an MLIB subprogram from within a C language program.

- Spell MLIB subprogram names in lowercase. For example, use `ddot` for DDOT.
- All arguments to MLIB subprograms are passed by address. This rule means that the only arguments you should pass to MLIB subprograms are pointers. If `x` is type `int`, `long`, `float`, `double`, or `float`, then `&x` is the address of `x`. Arrays and character strings already are passed as pointers and do not need the address-of operator, &. Because constants do not have addresses, use, for example, the declaration `int two = 2` and specify `&two` to pass the constant 2.

- By default, the first element of a Fortran array is `X(1)`, but the first element of a C array is `X[0]`. As a consequence, you must adjust any input or output arguments or function values that are indices into arrays by decrementing the quantity by 1. For example, because `idamax` returns the one-based index of the element of an array that has maximum magnitude, you might code

```
i = idamax_(&n, x,      ) - 1;
```

- MLIB expects arrays to be stored in column-major order; however, C stores them in row-major order. As a consequence, an MLIB subprogram sees the transpose of the C program's matrix. Three ways to handle this incompatibility are:

  - In your C program, use the transpose of the matrix you are using so it would appear to be in column-major order to the MLIB subprogram.
  - Some MLIB subprograms have a "transpose" option that uses the transpose of the matrix to do its calculations. In this case, store the matrix in normal order (for C) and use the transpose option to solve the intended problem. For example, to solve a system of linear equations $Ax = b$, factor the matrix you call A but which MLIB sees as A^T with DGEFA, and use job = 1 in DGESL to solve for x.
  - Recast the computation so that it operates on C's row-major order. For example, the matrix multiplication $C = AB$ can be recast as $C^T = B^T A^T$. Thus, for example, VECLIB subroutine DGEMM can be used with `transa = 'NonTransposed'` by reversing the order of the matrices A and B and their transposition options, sizes, and leading dimensions in the CALL statement.

- Another consequence of the difference in storage order of arrays is that "leading dimension" arguments, such as `lda` in subroutine DGEMV, must be defined based on the last dimension of the C array declaration. For example, the C declaration

```
double a[2][4];
```

  is equivalent to the Fortran declaration

```
REAL*8 a(4,2)
```

  and corresponds to `lda = 4`.

- MLIB subprograms that expect character arguments can be called from C with a string constant, a character array, or a pointer to a `char` variable for each character argument. In addition, an `int` must be passed at the end of the argument list for each character argument. These extra arguments are in the same order as the character arguments and indicate the length of the character string, not counting any "\0" terminator. For subroutine XERBLA, the quantity passed for the argument name must be at least six characters long, and only six characters are significant.

- Because C does not support co[illegible] arithmetic in a predefined way, storage layouts for Fortran types COM[illegible]3 and COMPLEX*16 usually are defined by typedef statements. Use th[illegible]owing typedef statements:

```
typedef struct {float  re, i    complex8_t;
     /* COMPLEX*8  type *
typedef struct {double re, i    complex16_t;
     /* COMPLEX*16 type *
```

- Using the above typedef sta[illegible]s, Fortran and C declarations are related as shown in Table A-1.

Table A-1 Relationship Between Fortra[illegible]d C Declarations

Fortran	C
LOGICAL*4 l	[illegible]t l;
INTEGER*4 i	[illegible]t i;
INTEGER*8 i	[illegible]ng long int i;
REAL*4 s	[illegible]oat s;
REAL*8 d	[illegible]uble d;
COMPLEX*8 c	[illegible]mplex8_t c;
COMPLEX*16 z	complex16_t z;
CHARACTER*(n) t	char t[n];
PROGRAM main	[illegible]main ()
SUBROUTINE sub (...)	[illegible]oid sub (...)
SUBROUTINE SUB (...)	[illegible]oid sub (...)
INTEGER*4 FUNCTION intf(...)	[illegible]t intf (...)
REAL*4 FUNCTION sf (...)	[illegible]oat sf(...)
REAL*8 FUNCTION df (...)	[illegible]uble df (...)
COMPLEX*8 FUNCTION cf (...)	[illegible]mplex8_t cf (...)
COMPLEX*16 FUNCTION zf (...	[illegible]mplex16_t zf(...)

Header files

C header files f VEC e installed in /opt/mlib/include as file veclib.h. Header file lap k.h is rovided for use with its associated libraries. The following discus on ap o all these header files. Include the -I/opt/mlib/ nclud on on your C compiler command line to access the C header files.

Use the #incl e dire o incorporate one of these header files into your program, for ex mple,

```
#include eclib.
```

Each header fil includ SI C function prototypes and PCC declarations for all subprogram Using eader files is optional but, if you do not use them, you must decla the re value types of the functions you use or C treats them as integer

If you need to use more one of the header files in a C program file, you can include them in any ord ing #include.

The header files ake ca many of the interlanguage programming rules listed in the pr ious se . Consider the following header file entry for VECLIB funct n DGEM

```
#ifndef  MM
#define  MM d
#define  emm_ d
extern v  d dgem    r* transa, char* transb, int* m, int* n,
          * k,      * alpha, double a[], int* lda,
          ble b,    t* ldb, double* beta, double c[],
          * ldc,    len_transa, int len_transb);
#endif
```

- The header file entry ach subprogram includes #define statements that remove an unde e to the subprogram name if you code one. If you code the sub rogram n e in upper case and omit the underscore, the name is translate to lowe . To use the header file but omit some of the function pr types, ine the upper case form of the name before the #include o the hea ile.
- The ANSI C unction totypes in the header files enforce call by address by explicitly eclarin ery argument that is not an array is to be passed by address.

- The ANSI C function prototype [illegible] enforce the additional `int` arguments at the end of the argument list [illegible] sponding to the character arguments. Note the two `char*` arguments at [illegible] e beginning of the argument list in the function prototype for DGEM[illegible] two `int` arguments at the end, which must tell the lengths of the [illegible] r strings `transa` and `transb`.
- The `typedef` statements that [illegible] given to define the two data types, `complex8_t` and `complex1[illegible]` included in the header file and used throughout wherever `COMPLEX` [illegible] `COMPLEX*16` variables or arrays are required.
- If you are compiling with the [illegible] ompiler option `-Ac`, the ANSI C prototypes are replaced by Ker[illegible] n-and-Ritchie-style declarations. For example, the prototype descri[illegible] ve for DGEMM is replaced with

```
extern void dgemm ();
```

Examples

Calling subroutines

This section gives an example of [illegible] VECLIB subprograms to solve a system of linear equations. This example [illegible] resents a Fortran code segment, then a corresponding C code segment. The [illegible] eration of the matrix and right-hand-side values is omitted from both examples.

The problem is to solve a system of [illegible] equations $Ax = b$, where A is a 6-by-6 matrix stored in a `double` (equiva[illegible] o type `REAL*8` in Fortran) array whose dimensions are 10 by 10, and b is [illegible] or 6 elements long stored in a `double` array of dimension 10.

Figure A-1 Calling VECLIB routines from Fortran and C

Fortran:

```
      PROGRAM MA
      INTEGER*4 I       , IER, JOB
      PARAMETER (       = 10 )
      INTEGER*4 I       DA)
      REAL*8    A       DA), B(LDA)
      N = 6
      JOB = 0   !       E COLUMN-MAJOR-STORED SYSTEM
      CALL DGEF         DA, N, IPVT, IER)
      IF ( IER .        ) THEN
           CALL D       A, LDA, N, IPVT, B, JOB)
      ELSE
          WRITE (       "singular matrix"
      END IF
      END
```

C:

```
#include <stdio.h
#include <veclib.
#define LDA 10
main ()
{
    int ld  = LD
    int ier;
    int n = 6;
    int j b = 1;        o solve transpose of
                        -major-stored system */
    int ipvt[LDA]
    double a[LDA]       ;
    double b[LDA]

    dgef  (a, &ld       , ipvt, &ier);
    if ( ier == 0
        sl (a,          , &n, ipvt, b, &job);
    else
        intf ("         ar matrix\n");
}
```

Calling functions

The method of calling a VECLIB function subprogram depends on its data type. The following examples show how to call some Fortran function subprograms from C. The function prototypes in clib.h cast the function return values to the proper data types.

REAL functions

The program in Figure A-2 computes the dot product of two `double` (Fortran `REAL*8`) vectors of length 2 using the VECLIB subprogram `DDOT`.

Figure A-2 **Calling a real-valued function from C**

```
#include <stdio.h>
#include <veclib.h>

main ()
{
    int one = 1;
    int two = 2;
    double s;
    static double x[2] =  1. , -2.0};
    static double y[2] = -3. , -2.0};

    s = ddot (&two, x, &one, y, &one);
    printf ("ddot = %f\n", s);
}
```

COMPLEX functions

Figure A-3 illustrates how to call a complex16_t (Fortran COMPLEX*16) VECLIB function subprogram ZDOTC.

Figure A-3 **Calling a complex-valued function from C**

```
#include <stdio.h>
#include <veclib.h>

main ()
{
    int one = 1;
    int two = 2;
    complex16_t s;
    static complex16_t x[]     {{-1.0, 2. },{ 2.0,-3.0} };
    static complex16_t y[]     {{ 3.0,-1. },{ 2.0, 1.0} };

    s = zdotc (&two, x, &one, y, &one);
    printf ("zdotc = (%e, %e)\n", s.re, s.im);
}
```

Linking VECLIB sub rograms into a C program

You can call VECLIB (APACK subprograms from a program coded in C by including -lveclib -lcl lapack -lcl on the cc command line that links your program:

cc -o *test test.c* -lvecli

cc -o *test test.c* -llap

Error handling

MLIB contains two st l error handlers, subroutines XERBLA and XERVEC. Refer to "XE \" on page 258 and "XERVEC" on page 526. In addition, the documen for every subprogram indicates if it calls an error handler and, if so, und t conditions.

Error handlers in VEC and LAPACK work the same way.

The error handlers are signed to be compatible with both the Fortran and the C runtime systems. If her error handler is called within a Fortran program, it writes an error mess e onto the standard error file and terminates execution with a nonzer exit status. When called within a C program, the error handler writes the erro nessage onto the standard error file and raises signal SIGIOT. If the program s not provided a signal handler for SIGIOT, a core image is produced and tion terminates with a nonzero exit status. This is illustrated in Figure A-

Figure A-4 **Default VECLIB error handling in a C program**

```
% cat  vsig1.c
#include <veclib.h>

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;

    sflr1c (&n, &alpha, x,    cx);
}% cc vsig1.c  -lveclib
% a.out
*****************************  ****************************************
* XERVEC: subprogram SFLR1C c   d with invalid value of argument number 4 *
*****************************  ****************************************
IOT trap (core dumped)
% echo $status
134
```

The fourth argument, incx, in the call to VECLIB subprogram SFLR1C must be nonzero. When SFLR1C detects that it is zero, it calls the error handler XERVEC, which writes the error message and raises signal SIGIOT. Because no signal handler has been enabled for this signal, the system writes a core image file and terminates the process.

NOTE

If the user has a signal handler that is enabled by the SIGIOT signal, that signal handler is executed if default error handling is in effect.

As a C programmer, you can alter this default behavior or change the signal used to any other valid signal (as given in signal(3c)), as shown in the next section.

Changing the error handler signal

VECLIB includes a C-callable function, initMLIB, that can be used to change the signal used by the standard error handlers to something other than SIGIOT. Usage is:

```
#include <signal.h>

int initMLIB(sig)
int sig;
```

Sig is 0 or one of the signal numbers or names as given in /usr/include/signal.h. The specified signal is used instead of the default signal, SIGIOT. None of the error handlers is called. If sig is 0, SIGIOT is used. In addition, a simple signal handler is installed that executes exit(1) to terminate the program with a nonzero exit status. This is the C equivalent to executing a FORTRAN STOP statement.

Figure A-5 illustrates the use of initMLIB, whose function prototype is obtained from veclib.h.

Figure A-5 **Changing the VECLIB error handling signal**

```
% cat sig2.c
#include <signal.h>
#include <veclib.h>

main ()
{
    int n = 1;
    float alpha = 2.0;
    float x[10];
    int incx = 0;

    initMLIB (SIGUSR1);
    sfill_ (&n, &alpha, x, &incx);
    printf("Hello world\n");
}
% cc sig2.c -lveclib
% a.out
******** ********** ***********************************************************
* XERVLA ** Subprogram SFILL_ called with invalid value of argument number 4 *
******** ********** ***********************************************************
% echo $status
1
```

In Figure A-5, the call to initMLIB changes VECLIB's error handling signal to SIGUSR1 and enables VECLIB's simple signal handler for this signal. If 0 or SIGIOT had been used in place of SIGUSR1, the only differences in behavior from the program in Figure A-5 would be the core image and the exit status.

NOTE

The initialization function for LAPACK also is called `initMLIB`.

Any previously established signal handler for signal `sig` is replaced by VECLIB's default signal handler.

Supplying your own signal handler

After calling `initMLIB` to specify [illegible] signal VECLIB's standard error handlers are to use, you can [illegible] your own signal handler for that signal via the C function *signal*(3c). This [illegible] illustrated by the program in Figure A-6.

Figure A-6 **Changing the error handling signal handler**

```
% cat  vsig3.c
#include <signal.h>
#include <veclib.h>

static void veclib_s[illegible]), scp)
    int sig, code;
    struct sigcontext [illegible]
{
    printf("SIGUSR2 h[illegible] ... exiting
    exit(123);
}

main ()
{
    int n = 1;
    float alpha = 0.[illegible];
    float x[10];
    int incx = 0;
    int oldsig;

    initMLIB (SIGUSR2);
    oldsig = (int)si[illegible] veclib_sig);
    sflr1c_ (&n, &alp[illegible] ;
}% cc vsig3.c  -lvecl[illegible]
% a.out
************************[illegible] ****************************************
* XERVEC: subprogram SF[illegible] ca[illegible] with invalid value of argument number 4 *
************************[illegible] ****************************************
SIGUSR2 has occurred...
% echo $statu[illegible]
123
```

In Figure A-6, the call to in[illegible] anges VECLIB's error handling signal to `SIGUSR2` and enables VECLIB's [illegible] signal handler for this signal. Subsequently, the signal ha[illegible] sig is enabled by the call to `signal`.

When VECLIB subprogram [illegible] tects the error in arguments, it calls error handler XERVEC, whi[illegible] the error message and raises signal `SIGUSR2`. Finally, the syste[illegible] eclib_sig.

Error handling

B LINPACK Subprogr ms

Overview

LINPACK is a collection of Fortra ib utines that analyze and solve systems of simultaneous linear algebraic e ns. The LINPACK package was develo d with the support of th l Science Foundation in the latter part of the 1970s as one of the fir n etely systemized collections of linear algebra software.

Since that time, much has change mputer architectures and mathematical algorithms, and LA K supersedes LINPACK.

Except for two subprograms, LIN s not included in HP MLIB (version 6.1 or later). This appendix describes o subprograms included in HP VECLIB:

- DGEFA—A subprogram to fac eral matrix
- DG SL—A subprogram to so equations with a general matrix

The following manual provides co documentation for LINPACK:

Dongarra, J.J., J.R. Bunch, C.B. M d G.W. Stewart. *LINPACK Users' Guide*. Philadelphia: Society for I l and Applied Mathematics, 1979.

LINPACK subprogram included in VECLIB

Name DGEFA
LU factorization

Purpose This subprogram computes the triangular factorization of a general dense n-by-n matrix A. Specifically, given A, DGEFA determines an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and an n-by-n upper-triangular matrix U such that

$$PA = LU.$$

Computational singularity results in one or more zero diagonal elements of U. This condition is detected during factorization, and a status response is returned to indicate its occurrence.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = Pb$. This operation is performed by the VECLIB subprogram DGESL. Refer to "DGESL" on page 584 for more information.

Usage VECLIB:

```
INTEGER*4    lda, n, ipvt(n), ier
REAL*8       a(lda, n)
CALL DGEFA(a, lda, n, ipvt, ier)
```

Input

a Array containing the **n**-by-**n** matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Output

a The triangular factors replace the input matrix; the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U. **a** must be preserved between the factorization call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P. **ipvt** must be preserved between the factorization call and any solve, determinant, or inverse call.

ier	Status response:	
	ier = 0	Normal return.
	ier = $k \neq 0$	If $u_{kk} = 0$. (u_{kk} is the ***k*-th element on** the diagonal of upper triangular matrix *U*). Technically, this is not an error condition for these subprograms, but it does indicate that ***A*** is computationally singular and that a division by zero occurs if the factorization is used to solve a system of linear equations.

Notes

DGEFA is usage compatible with the standard LINPACK subprogram.

The triangular factors are stored in a different format from the format used by the standard LINPACK subprograms. They are compatible with the VECLIB subprogram DGESL.

Example

Factor the 6-by-6 REAL*8 matrix A stored in array A whose dimensions are 10-by-10.

```
INTEGER*4 LDA,N,IPVT( 
REAL*8    A(10,10)
LDA = 10
N = 6
CALL DGEFA (A,LDA,N,
IF ( IER .NE. 0 ) TH
    handle singular 
END IF
```

Name

DGESL
Solve linear equations

Purpose

Given the triangular fac tion of a general dense n-by-n coefficient matrix A, and a right-hand-si or b, this subprogram solves the system of linear equations $Ax = b$. Spe given an n-by-n permutation matrix P, an n-by-n unit lower-triang atrix L, and a nonsingular n-by-n upper-triangular matrix ch that

$$PA = LU,$$

and an n-vector b, to f isfying $Ax = b$, the subprograms compute

$$v = Pb,$$

then successively solv

$$Lw = v$$

and

$$Ux = w.$$

To solve $A^*x = b$, the ams successively solve

$$U^*v = b$$

and

$$L^*w = v,$$

and then compute

$$x = P^*w.$$

The triangular factors o efficient matrix may be computed by the companion subprogram A. This computes only the factorization, using an elementary test for sing of the coefficient matrix. Refer to "DGEFA" on page 582 for more infor

Usage

VECLIB:

INTEGER*4 , ipvt(n), job
REAL*8 n), b(n)
CALL DGESL(a, ld pvt, b, job)

Input

a	Array containing the triangular factors L and U of the **n**-by-**n** coefficient matrix A as computed by the companion factorization or condition number estimation subprogram. **a** must have been preserved between the factorization or condition number call and the solve call.
lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq \max(\mathbf{n},1)$.
n	The order of matrix A, **n** ≥ 0.
ipvt	The pivot information necessary to construct the permutation matrix P as computed by the companion factorization condition number estimation subprograms. **ipvt** must have been preserved between the factorization or condition number call and the solve call.
b	The right-hand-side vector b.
job	Option flag **job** = 0 Solve $Ax = b$. **job** $\neq$ 0 Solve $A^*x = b$.

Output

b	The solution vector x overwrites the right-hand-side vector b.

Notes DGESL is usage compatible with LINPACK and LINPACK subprograms.

Example Solve a system of linear equations $Ax = b$, where A is a 6-by-6 REAL*8 matrix stored in array A whose dimensions are 10-by-10, and b is a vector 6 elements long stored in an array B of dimension 10.

```
INTEGER*4 LDA,N,IPVT(10),JOB,IER
REAL*8    A(10,10),B(10)
LDA = 10
N = 6
JOB = 1
CALL DGEFA (A,LDA,N,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
    CALL DGESL (A,LDA,N,IPVT,B,JOB)
ELSE
    Handle singular case
END IF
```

If the coefficient matrix A is determined to be nonsingular, the solution vector x overwrites the right-hand-side b in array B.

C Parallelized Subprograms

Overview

Many subprograms in HP MLIB are parallelized to improve performance on Hewlett-Packard computers. This appendix lists parallelized routines in both HP VECLIB and HP LAPACK. Refer to Table C-1 on page 588 for VECLIB subprograms, and Table C-2 on page 589 for LAPACK subprograms. Many additional subprograms in the library call the BLAS and so inherit BLAS parallelism.

Refer to "Parallel processing" on page 533 and "Parallel processing" on page 534, for information about enabling parallel processing in VECLIB and LAPACK, respectively.

Parallelized subprograms in VECLIB

Table C-1 lists the HP MLIB VECLIB subprograms that are parallelized to improve performance.

Table C-1 Parallelized subprograms in VECLIB

Subroutine	Description
{S,D,C,Z}GEMM	General Matrix-Matrix Multiply
{S,D,C,Z}GEMV	General Matrix-Vector Multiply
{S,D}GER	General Rank-1 Update
{C,Z}GERC	General Rank-1 Update
{C,Z}GERU	General Rank-1 Update
{S,D}GBFA	Factor a General Band Matrix
DSLEFA	Numeric Factor Sparse Matrix
DSLEFS	Solve Sparse Linear Equations
{S,D,C,Z}1DFFT	One-Dimensional FFT
{S,D,C,Z}2DFFT	Two-Dimensional FFT
{S,D,C,Z}3DFFT	Three-Dimensional FFT
{S,D,C,Z}FFTS	Simultaneous Complex-to-Complex One-Dimensional FFT
{S,D,C,Z}RC1FT	Real-to-Complex One-Dimensional FFT
{S,D,C,Z}RC2FT	Real-to-Complex Two-Dimensional FFT
{S,D,C,Z}RC3FT	Real-to-Complex Three-Dimensional FFT
{S,D,C,Z}RCFTS	Simultaneous Real-to-Complex One-Dimensional FFT
CONV	Correlation, Convolution

Parallelized subprograms in LAPACK

Table C-2 lists the HP LAPACK subprograms that are parallelized to improve performance.

Table C-2 Parallelized subprograms in LAPACK

Subroutine	Description
{S,D,C,Z}GBTRF	Factor a General Band Matrix
{S,D,C,Z}GBSV	Solve a General Band Linear System
{S,D,C,Z}GESV	Solve a General Full System
{S,D,C,Z}GBSVX	Solve a General Band Linear System
{S,D,C,Z}GESVX	Solve a General Full Linear System
{S,D,C,Z}GETRF	Factor a General Full Matrix
{S,D,C,Z}GETRI	Invert General Matrix
{S,D,C,Z}GETRS	Solve a General Full Linear System
{S,D}GELS	Solve General Least Squares Problem
{S,D}GELSS	Solve General Least Squares Problem
{S,D}GEQRF	QR Factorization of General Matrix
{S,D}GGQRF	Generalized QR Factorization
{S,D}GGRQF	Generalized RQ Factorization
{S,D}GEGS	Generalized Schur Form
{S,D}GEGV	Generalized Generalized Eigenproblem
{S,D}GESVD	Singular Value Decomposition
{S,D,C,Z}GEMM	General Matrix-Matrix Multiply
{S,D,C,Z}GEMV	General Matrix-Vector Multiply
{S,D}GER	General Rank-1 Update
{C,Z}GERU	General Rank-1 Update
{C,Z}GERC	General Rank-1 Update

Parallelized subprograms in LAPACK

Index

Symbols

A

B

C

D

E

G

J

L

M

N

O

P

R

S

Z