HP MLIB User's Guide

Seventh Edition

HP Part No. B6061-96027 and B6061-96028
HP MLIB
December 2004

Printed in USA

Edition: Seventh

Document Numbers: B6061-96027 and B6061-96028
Remarks: Released Dcember 2004 with HP MLIB software version9.0. User's Guide split into two volumes with this release.

Edition: Sixth

Document Number: B6061-96023
Remarks: Released September 2003 with HP MLIB software version 8.50.

Edition: Fifth

Document Number: B6061-96020
Remarks: Released September 2002 with HP MLIB software version 8.30.

Edition: Fourth

Document Number: B6061-96017
Remarks: Released June 2002 with HP MLIB software version 8.20.

Edition: Third

Document Number: B6061-96015
Remarks: Released September 2001 with HP MLIB software version 8.10.

Edition: Second

Document Number: B6061-96012
Remarks: Released June 2001 with HP MLIB software version 8.00.

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

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Table of Contents

Figures

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 systems ranging from single-processor workstations to multiprocessor high-end servers. HP MLIB is optimized for HP PA-RISC 2.0, Itanium™ 2, and Opteron processors. HP MLIB has six components; VECLIB, LAPACK, ScaLAPACK, SuperLU, SOLVERS, and VMATH.

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 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/index..html

ScaLAPACK

ScaLAPACK is a library of high-performance linear algebra routines capable of solving systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. ScaLAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

ScaLAPACK is a public domain software that was developed by Oak Ridge National Laboratory. It is designed for distributed computing and uses the Message Passing Interface (MPI) for parallelism. This implementation provides a version of ScaLAPACK tuned on HP servers and built with HP's MPI. The ScaLAPACK library routines are written in Fortran 77 and are callable from Fortran 90 and C routines. Unlike other MLIB libraries, there is not a version of ScaLAPACK that assumes all integers are 8 bytes in length.

Refer to Part 3 of this manual for information specific to HP ScaLAPACK. To supplement the HP specific information provided in Part 3 of this document, refer to the standard *ScaLAPACK Users' Guide*. You can access the latest edition of the *ScaLAPACK Users' Guide* at the Netlib repository at the following URL:

http://www.netlib.org/scalapack/slug/index..html

SuperLU

This implementation provides the Distributed SuperLU library designed for distributed memory parallel computers. It was based on the public-domain SuperLU_DIST, which was developed at the Lawrence Berkeley National Lab and the University of California at Berkeley.

The library contains a set of subroutines to solve a sparse linear system. The library is implemented in ANSI C, using HP Message Passing Interface (MPI) for communication. The library includes routines to handle both real and complex matrices using double precision. The parallel routine names for the double-precision real version start with the letters "pd" (e.g. **pdgstrf**). The parallel routine names for the double-precision complex version start with letters "pz" (e.g. **pzgstrf**). Unlike other MLIB libraries, there is not a version of SuperLU that assumes all integers are 8 bytes in length.

The routines can be called directly from C applications. They may also be called from Fortran applications, however, "bridge" routines (in C) must be supplied. For details about creating bridge routines, please refer to Section 2.9.2 in the *SuperLU User's Guide* available at:

http://www.nersc.gov/~xiaoye/SuperLU

SOLVERS

Solvers is a collection of direct sparse linear system solvers and graph partitioning routines. Symmetric systems can be solved using SMP parallelism and structurally-symmetric systems can be solved using out-of-core functionality. These routines have been optimized for use on Hewlett-Packard servers. Features are:

- Sparse symmetric and structurally-symmetric linear equation solutions.
- Sparse symmetric ordinary and generalized eigensystem solutions.
- Out-of-core symmetric and structurally-symmetric linear equation and eigensystems solutions.
- Full METIS functionality

  This implementation provides the METIS Version 4.0.1 library. It is based on the public-domain METIS, which was developed at the University of Minnesota, Department of Computer Science, and the Army HPC Research Center. The library contains a set of subroutines for graph partitioning, mesh partitioning, and sparse matrix reordering, as well as auxiliary routines. HP MLIB contains the full METIS functionality as that in the public domain METIS, however, the routine names are different. HP MLIB METIS routine names have been prepended with `mlib_` to avoid name conflict on applications and libraries that contain their own local version of METIS.

  For more information about METIS, please refer to:

  http://www-users.cs.umn.edu/~karpis/metis/metis/index.html

VMATH

VMATH is a library of vector math routines corresponding to many of the widely used scalar math routines available with C, C++, and Fortran90.

VMATH is intended for computationally intensive mathematical applications amenable to a vector programming style.

VMATH provides two libraries: VMATH, whose interface uses 4-byte integers; and VMATH8, whose interface uses 8-byte integers and is otherwise equivalent to VMATH. VMATH routines come with both Fortran and C interfaces.

For more detailed information on VMATH as well as subprogram specifications, please refer to the VMATH chapter in Part 5 of this book. The VMATH man pages provide a man page for each subprogram.

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, HP MLIB LAPACK in Part 2, HP MLIB ScaLAPACK in Part 3, and HP MLIB Distributed SuperLU in Part 4.

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 describes sparse BLAS operations
- Chapter 5 describes the discrete Fourier transforms in VECLIB
- Chapter 6 describes subprograms to compute convolutions and correlations of data sets
- Chapter 7 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 8 describes information specific to Hewlett-Packard's implementation of LAPACK
- Chapter 9 describes selected LAPACK auxiliary subprograms

Part 3 of this document is organized as follows:

- Chapter 10 describes ScaLAPACK functionality

Part 4 of this document is organized as follows:

- Chapter 11 describes Distributed SuperLU functionality

Part 5 of this document is organized as follows:

- Chapter 12 describes VMATH functionality

Part 6 of this document is organized as follows:

- Chapter 13 explains sparse symmetric linear equation subprograms
- Chapter 14 describes METIS subprograms
- Chapter 15 describes sparse symmetric eigenvalue subprograms
- Chapter 16 describes BCSLIB-EXT functionality

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 *LAPACK Users' Guide* is a publication from the Society for Industrial and Applied Mathematics that 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

Supplemental information for Part 3 of this *HP MLIB User's Guide*, is found in the *ScaLAPACK User's Guide*.

The *ScaLAPACK Users' Guide* is a publication from the Society for Industrial and Applied Mathematics that provides an informal introduction to the design of ScaLAPACK as well as a detailed description of its contents and a reference manual. You can access the latest edition of the *ScaLAPACK Users' Guide* at the Netlib repository at the following URL:

http://www.netlib.org/scalapack/slug

Supplemental information for Part 4 of this *HP MLIB User's Guide*, is found in the *SuperLU User's Guide*, which is only available online at:

http://www.nersc.gov/~xiaoye/SuperLU

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 DAXPY**.

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 DAXPY (n, a, x, incx, y, incy)

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

The *HP MLIB User's Guide*, the *LAPACK Users' Guide,* and the *ScaLAPACK 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

For the latest edition of the *ScaLAPACK User's Guide*, refer to:
http://www.netlib.org/scalapack/slug

For the latest edition of the *SuperLU User's Guide*, refer to:
http://www.nersc.gov/~xiaoye/SuperLU

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.
- Sca*LAPACK Users' Guide* Philadelphia, PA: Society for Industrial and Applied Mathematics. This guide provides information on the subprograms provided with the ScaLAPACK library.
- *Parallel Programming Guide for HP-UX Systems*. Describes efficient shared-memory parallel programming techniques using HP compilers.
- *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 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.)
- *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.
- *Software Optimization for High Performance Computing: Creating Faster Applications:* Provides state-of-the-art solutions for every key aspect of software performance; both code-based and algorithm-based.
- *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.
- *IA-64 and Elementary Functions*. Describes the architecture of the IA-64 processor and how to implement elementary functions.
- METIS documentation is available online at http://www-users.cs.umn.edu/~karypis/metis/metis/index.html.
- OpenMP documentation is available online at http://www.openmp.org.

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 Level 1 BLAS
- Sparse vector and matrix operations, including the Sparse BLAS
  - BLAS 3 routine DGEMM is highly tuned
- Matrix operations, including the Level 2 and Level 3 BLAS
- CXML Blas extenstions

  Compaq Extended Math Library (CXML) is a collection of routines that performs numerically intensive operations that occur frequently in engineering and scientific computing, such as linear algebra and signal processing.

  HP MLIB adds support for the unique CXML extensions to the legacy BLAS and for the array math functions in CXML. However, the additional support of XCML is not exhaustive. For more information about CXML see the *CXML Reference Manual* part number AA-PV6VE-TE.
- Discrete Fourier transforms
- Convolution and correlation
- Miscellaneous tasks, such as sorting and generating random numbers

VECLIB provides two sets of libraries: VECLIB and VECLIB8. To determine if a subprogram is included in VECLIB8, refer to the VECLIB8 section under each subprogram specification in the following chapters.

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
- 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
- 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.

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 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 and VECLIB8 libraries are available as archive and shared libraries. They consist 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 or VECLIB8 be used as an object library at link time. Refer to the LAPACK chapter for details about accessing the LAPACK library, The ScaLAPACK chapter for details about accessing the ScaLAPACK library, and the Distributed SuperLU chapter for details about accessing the Distributed SuperLU library.

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

Table 1-1 VECLIB and VECLIB8 Libraries

Processor Type	OS Version	Address Width	Installation Directory	Libraries
PA 2.0	HP-UX 11i or later	32-bit	/opt/mlib/lib/pa2.0	libveclib.a libveclib.sl
		64-bit	/opt/mlib/lib/pa20_64	libveclib.a libveclib.sl libveclib8.a libveclib8.sl
Itanium 2	HP-UX 11i V1.6 or later	32-bit	/opt/mlib/lib/hpux32	libveclib.a libveclib.so
		64-bit	/opt/mlib/lib/hpux64	libveclib.a libveclib.so libveclib8.a libveclib8.so
Itanium 2	Red Hat Linux Enterprise 3 Intel V8.0 Compiler	64-bit	/opt/mlib/intel_8.0/hpmpi_2.1/lib/64	libveclib.a libveclib.so libveclib8.a libveclib8.so
Itanium 2	Red Hat Linux Enterprise 3 Intel IA-32 V8.0 Compiler	32-bit	/opt/mlib/intel_8.0/hpmpi_2.1/lib/32	libveclib.a libveclib.so libveclib8.a libveclib8.so
Itanium 2	HP XC6000 Intel V8.0 Compiler	64-bit	/opt/mlib/intel_8.0/hpmpi_2.1/lib/64	libveclib.a libveclib.so libveclib8.a libveclib8.so

Processor Type	OS Version	Address Width	Installation Directory	Libraries
Itanium 2	HP XC6000 Intel V7.1 Compiler	64-bit	/opt/mlib/intel_7.1/hpmpi_2.1/lib/64	libveclib.a libveclib.so libveclib8.a libveclib8.so
Itanium 2	HP XC4000 PGI V5.1 Compiler	64-bit	/opt/mlib/intel_5.1/hpmpi_2.1/lib/64	libveclib.a libveclib.so libveclib8.a libveclib8.so
Itanium 2	Opteron PGI V5.1 Compiler	64-bit	/opt/mlib/intel_5.1/hpmpi_2.1/lib/64	libveclib.a libveclib.so libveclib8.a libveclib8.so

Only 64-bit addressing is supported in the VECLIB8 library.

The file name of the archive VECLIB and VECLIB8 libraries are libveclib.a and libveclib8.a respectively. The VECLIB shared library for PA is libveclib.sl and is libveclib.so for all other operating systems.

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. See “Linking with libisamstub.a” on page 8 if your application calls Fortran77 routines from C language code, and you are linking with the VECLIB archive library.

The HP VECLIB library contains parallelized subprograms. Refer to Appendix C, “Parallelized Subprograms,” for details. If your program is run on an HP-UX system with multiple processors, see “Parallel processing” on page 17 for additional information about linking your program.

Compiling and linking (VECLIB)

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 **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.

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**.

When you use the **–aarchive_shared** flag on your compiler command line for HP-UX, it ensures that the compiler links the archive library. If the archive library is not available, then it links the shared library. If you omit **–aarchive_shared** and **–ashared_archive**, the linker defaults to linking the shared library. Link with **–Bstatic** on Linux systems.

If your program uses subprograms from VECLIB, LAPACK, ScaLAPACK, Distributed SuperLU, SOLVERS and VMATH, specify –**llapack**, –**lveclib**, –**lscalapack**, –**lsuperlu_dist**, –**lsolvers**, and –**lvmath** or –**llapack8**, –**lveclib8,** –**lscalapack8**, –**lsuperlu_dist8**, –**lsolvers8** and –**lvmath8** on the compiler command line.

NOTE Do not mix subprograms from the two types of 64-bit address libraries (those with 32-bit integers and those with 64-bit integers) in the same program.

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.

For PA-Based Systems

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 –lm**

   aCC [*options*] *file ...* **–Wl,–aarchive_shared –lveclib –lcl –lm**

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- or 64-bit addressing on a PA-based system, use one of the following:

   f90 [*options*] *file ...* **/opt/mlib/lib/[pa2.0|pa20_64]/libveclib.a**

   cc [*options*] *file ...* **/opt/mlib/lib/[pa2.0|pa20_64]/libveclib.a –lcl –lm**

   aCC [*options*] *file ...* **/opt/mlib/lib/[pa2.0|pa20_64]/libveclib.a –lcl –lm**

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

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

   –Wl,–aarchive_shared,–L/opt/mlib/lib/[pa2.0|pa20_64]

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

f90 [*options*] *file* ... **–Wl,–aarchive_shared,–L/opt/mlib/lib/[pa2.0|pa20_64] –lveclib**

cc [*options*] *file* ... **–Wl,–aarchive_shared,–L/opt/mlib/lib/[pa2.0|pa20_64] –lveclib –lcl –lm**

aCC [*options*] *file* ... **–Wl,–aarchive_shared,–L/opt/mlib/lib/[pa2.0|pa20_64] –lveclib –lcl –lm**

4. Set the LDOPTS environment variable to include:

   –aarchive_shared,–L/opt/mlib/lib/[pa2.0|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 –lm**

   aCC [*options*] *file* ... **–lveclib –lcl –lm**

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.

5. To link with the VECLIB8 libraries on a PA-based system, use the **+DA2.0W** option to specify memory addresses are 64-bit.

   Compile Fortran applications with the **+i8**, **+autodbl**, or **+autodbl4** option where:

   - **+i8** promotes 4-byte integer of logical constants, intrinsics, and user variables (declared as integer or logical) to 8-byte quantities.
   - **+autodbl** promotes all integer, logical, and real items to 8 bytes, and all double-precision and complex items to 16 bytes.
   - **+autodbl4** promotes all integer logical, and real items to 8 bytes, and complex items to 16 bytes. The **+autodbl4** option does not promote the size of double-precision and double-complex items.

   f90 +DA2.0W +i8 [*options*] *file* ... **–Wl,–aarchive_shared –lveclib8**

   cc +DA2.0W [*options*] *file* ... **–Wl,–aarchive_shared –lveclib8 –lcl –lm**

   aCC +DA2.0W [*options*] *file* ... **–Wl,–aarchive_shared –lveclib8 –lcl –lm**

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 –lm**

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

This option is only valid for 32-bit PA systems.

For Itanium-Based HP-UX Systems

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 –lm**

   aCC [*options*] *file ...* **–Wl,–aarchive_shared –lveclib –lcl –lm**

2. Specify the entire path of the library file on the compiler command line that links your program. To link your program with VECLIB for use with 32- or 64-bit addressing on an HP-UX system, use one of the following:

   f90 [*options*] *file ...* **/opt/mlib/lib/[hpux32|hpux64]/libveclib.a**

   cc [*options*] *file ...* **/opt/mlib/lib/[hpux32|hpux64]/libveclib.a –lcl –lm**

   aCC [*options*] *file ...* **/opt/mlib/lib/[hpux32|hpux64]/libveclib.a –lcl –lm**

   Replace **libveclib.a** with **libveclib.so** on your compiler command line if you want to link the shared library on an Itanium-based system.

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

   –Wl,–aarchive_shared,–L/opt/mlib/lib/[hpux32|hpux64]

   For example, the command lines in Method 2 for HP-UX could be written:

   f90 [+DD32|+DD64] [*options*] *file...*–**Wl,–aarchive_shared,–L/opt/mlib/lib/[hpux32|hpux64] –lveclib**

   cc [+DD32|+DD64] [*options*] *file ...* –**Wl,–aarchive_shared,–L/opt/mlib/lib/[hpux32|hpux64] –lveclib –lcl –lm**

   aCC [+DD32|+DD64] [*options*] *file ...* –**Wl,–aarchive_shared,–L/opt/mlib/lib/[hpux32|hpux64] –lveclib –lcl –lm**

4. Set the LDOPTS environment variable to include:

–aarchive_shared,–L/opt/mlib/lib/[hpux32|hpux64]

For example:

setenv LDOPTS "–aarchive_shared,–L/opt/mlib/lib/hpux32"

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

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

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

aCC [*options*] *file* ... –**lveclib** –**lcl** –**lm**

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.

5. To link with the VECLIB8 libraries on an HP-UX system, use the **+DD64** option to specify memory addresses are 64-bit.

   Compile Fortran applications with the **+i8**, **+autodbl**, or **+autodbl4** option where:

   - **+i8** promotes 4-byte integer of logical constants, intrinsics, and user variables (declared as integer or logical) to 8-byte quantities.
   - **+autodbl** promotes all integer, logical, and real items to 8 bytes, and all double-precision and complex items to 16 bytes.
   - **+autodbl4** promotes all integer logical, and real items to 8 bytes, and complex items to 16 bytes. The **+autodbl4** option does not promote the size of double-precision and double-complex items.

   f90 +DD64 +i8 [*options*] *file* ... –**Wl,–aarchive_shared** –**lveclib8**

   cc +DD64 [*options*] *file* ... –**Wl,–aarchive_shared** –**lveclib8** –**lcl** –**lm**

   aCC +DD64 [*options*] *file* ... –**Wl,–aarchive_shared** –**lveclib8** –**lcl** –**lm**

For Itanium-Based Red Hat Linux Systems

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

   ifort [*options*] *file* ... **–Wl,–Bstatic** –**lveclib** –**openmp**

   icc [*options*] *file* ... **–Wl,–Bstatic** –**lveclib** –**openmp**

   Link with **`-Bdynamic`** if you want the archive library on a linux system.

NOTE When you use the Intel V8.0 C compiler to link the SOLVERS library, you may require one or more of **-lifcore**, **-lifport**, or **-ldl**.

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- or 64-bit addressing on a Linux system, use one of the following:

   ifort [*options*] *file* ... **/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]/libveclib.a –openmp**

   icc [*options*] *file* ... **/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]/libveclib.a –openmp**

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

   When you use the C compiler for linking, you may require one or more of **-1CEPCF90**, **-1F90**, **-1IEPCF90**, **-1PEPCF90**, or **-1POSF90**.

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

   –Wl,–Bstatic,–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]

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

   ifort [*options*] *file*...**–Wl,–Bstatic,–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64] –lveclib –openmp**

   icc [*options*] *file* ... **–Wl,–Bstatic,–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64] –lveclib –openmp**

   When you use the C compiler for linking, you may require one or more of **-1CEPCF90**, **-1F90**, **-1IEPCF90**, **-1PEPCF90**, or **-1POSF90**.

4. Set the LDOPTS environment variable to include:

   –Bstatic,–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]

   For example:

   setenv LDOPTS "–Bstatic,–L/opt/mlib/lib/linux"

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

   ifort [*options*] *file* ... **–lveclib –openmp**

   icc [*options*] *file* ... **–lveclib –openmp**

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.

5. Use the following commands to link your programs for use with the VECLIB8 library on a Red Hat Linux system.

   ifort –**i8** [*options*] *file* ... **-L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]/lveclib8** –**openmp**

   icc [*options*] *file* ... –**L/opt/mlib/intel_8.0/hpmpi_2.1/lib/[32|64]/lveclib8** –**openmp**

   When you use the C compiler for linking, you may require one or more of **-lCEPCF90**, **-lF90**, **-lIEPCF90**, **-lPEPCF90**, or **-lPOSF90**.

For XC6000 Systems with the Intel V8.0 Compiler

1. Use one of the following compile line commands to link VECLIB:

   ifort [*options*] *file* ... **–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

   icc [*options*] *file* ... **–L/opt/mlib/intel_8.0/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

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 64-bit addressing on an XC6000 system with the Intel V8.0 compiler, use one of the following:

   ifort [*options*] *file* ... **/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64/libveclib.a** –**openmp**

   icc [*options*] *file* ... **/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64/libveclib.a** –**openmp**

   Replace **libveclib.a** with **libveclib.so** on your compiler command line if you want to link the shared library on an XC6000 system.

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

   –**Wl,–Bstatic,–L/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64**

   For example, the command lines in Method 2 for XC6000 could be written:

   ifort [*options*] *file*...–**Wl,–Bstatic,–L/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

   icc [*options*] *file* ... –**Wl,–Bstatic,–L/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

4. Set the LDOPTS environment variable to include:

   –Bstatic,–L/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64

For example:

setenv LDOPTS "–Bstatic,–L/opt/mlib/lib/intel_8.0/hpmpi_2.1/lib/64"

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

ifort [*options*] *file* ... –**lveclib** –**openmp**

icc [*options*] *file* ... –**lveclib** –**openmp**

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.

For XC6000 Systems with the Intel V7.1 Compiler

1. Use one of the following compile line commands to link VECLIB:

   efc [*options*] *file* ... **–L/opt/mlib/intel_7.1/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

   icc [*options*] *file* ... **–L/opt/mlib/intel_7.1/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

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 64-bit addressing on an XC6000 system with the Intel V7.1 compiler, use one of the following:

   efc [*options*] *file* ... **/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64/libveclib.a** –**openmp**

   icc [*options*] *file* ... **/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64/libveclib.a** –**openmp**

   Replace **libveclib.a** with **libveclib.so** on your compiler command line if you want to link the shared library on an XC6000 system.

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

   –**Wl,–Bstatic,**–**L/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64**

   For example, the command lines in Method 2 for XC6000 could be written:

   efc [*options*] *file*...–**Wl,–Bstatic,**–**L/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

   icc [*options*] *file* ...–**Wl,–Bstatic,**–**L/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64** –**lveclib** –**openmp**

4. Set the LDOPTS environment variable to include:

–Bstatic,–L/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64

For example:

setenv LDOPTS "–Bstatic,–L/opt/mlib/lib/intel_7.1/hpmpi_2.1/lib/64"

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

efc [*options*] *file* ... **–lveclib –openmp**

icc [*options*] *file* ... **–lveclib –openmp**

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.

For XC4000 Systems with the PGI V5.1 Compiler

1. Use one of the following compile line commands to link VECLIB:

   pgf90 [*options*] *file* ... **–L/opt/mlib/pgi_5.1/hpmpi_2.1/lib/64 –lveclib –mp**

   pgcc [*options*] *file* ... **–L/opt/mlib/pgi_5.1/hpmpi_2.1/lib/64 –lveclib –mp –lpgf90 –lpgf90_rpml –lpgf902 –lpgf90rtl –lpgftnrtl**

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 64-bit addressing on an XC4000 system with the PGI V5.1 compiler, use one of the following:

   pgf90 [*options*] *file* ... **/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64/libveclib.a –mp**

   pgcc [*options*] *file* ... **/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64/libveclib.a –mp –lpgf90 –lpgf90_rpml –lpgf902 –lpgf90rtl –lpgftnrtl**

   Replace **libveclib.a** with **libveclib.so** on your compiler command line if you want to link the shared library on an XC4000 system.

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

   –Wl,–Bstatic,–L/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64

   For example, the command lines in Method 2 for XC4000 could be written:

   pgf90 [*options*] *file*...**–Wl,–Bstatic,–L/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64 –lveclib –mp**

   pgcc [*options*] *file* ...**–Wl,–Bstatic,–L/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64 –lveclib –mp –lpgf90 –lpgf90_rpml –lpgf902 –lpgf90rtl –lpgftnrtl**

4. Set the LDOPTS environment variable to include:

   –Bstatic,–L/opt/mlib/lib/pgi_5.1/hpmpi_2.1/lib/64

   For example:

   setenv LDOPTS "–Bstatic,–L/opt/mlib/lib/pgi5.1/hpmpi_2.1/lib/64"

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

   pgf90 [*options*] *file* ... **–lveclib –mp**

   pgcc [*options*] *file* ... **–lveclib –mp –lpgf90 –lpgf90_rpml –lpgf902 –lpgf90rtl –lpgftnrtl**

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.

Problem with `+ppu` compatibility and duplicated symbols

All MLIB subprograms documented in the *HP MLIB User's Guide* have two entry points: one is compatible with the Fortran compiler option `+noppu` (no postpend underbar) and the second is compatible with the Fortran compiler option `+ppu` (postpend underbar). For example, the MLIB BLAS error handler XERBLA has entry points xerbla and xerbla_. Table 1-1 shows the Fortran 90 compiler defaults.

Table 1-2 Compiler Defaults

	32-bit Addressing	64-bit Addressing
PA-RISC	+noppu	+ppu
Itanium	+ppu	+ppu

Different compiler defaults can cause duplicate symbol definitions at link time as the following examples illustrate.

PA-RISC processors

Suppose you write your own Fortran version of the XERBLA subroutine. You compile on PA-RISC with `+DA2.0W` and the compiler turns on `+ppu` (See Table 1-1), so your xerbla.o object file has an entry point xerbla_. However, the PA-RISC MLIB libraries were compiled with `+noppu` and the entry points with underbars were added as synonyms to the entry points without underbars. Hence, the various MLIB subprograms that call XERBLA reference them as xerbla (without the postpended underbar). Therefore, your xerbla_ does not satisfy their CALL XERBLA, so the linker accesses xerbla.o from the library.

This file includes both entry points xerbla and xerbla_, and xerbla_ conflicts with your XERBLA subroutine.

Use one of the following workarounds on PA-RISC if you are compiling your own Fortran version of XERBLA:

- Compile your XERBLA with +noppu. This implies that you also compile any of your program that calls XERBLA with +noppu, and that you apply this rule recursively.
- Use f90 ALIAS directives:

  ```
  % !$HP$ ALIAS xerbla='xerbla'
  ```

  This prevents the compiler from postpending the underbar onto the entry point and external references for XERBLA.

This problem is not confined to the XERBLA subroutine. It occurs on PA-RISC when the following conditions are met:

- Any subprogram with the same name as a user-visible MLIB subprogram is compiled in your program with +ppu
- Another MLIB subprogram used by your program also calls that subprogram

Furthermore, the reverse can occur in the PA-RISC 32-bit libraries if you compile with the +ppu compiler option.

Itanium processors

If you compile on Itanium using the compiler option +noppu your xerbla.o object has an entry point xerbla (without the postpended underbar). However the Itanium MLIB libraries were compiled with +ppu option (following the default behavior described in Table 1-1) and the entry points without underbars were added as synonyms to the entry points with underbars. Hence the various MLIB Fortran source subprograms that call XERBLA reference them as xerbla_. Therefore, your xerbla (without the postpended underbar) does not satisfy the Fortran calls to XERBLA, so the linker accesses xerbla.o from the library. This file contains both entry points xerbla and xerbla_, and xerbla conflicts with your XERBLA subroutine.

Moreover, the various MLIB C source subprograms that call XERBLA still reference them as xerbla (without the postpended underbar). It means that both xerbla and xerbla_ entry points must be provided in your xerbla.o.

Use one of the following workarounds on Itanium if you are compiling your own Fortran version of XERBLA:

- Compile your Fortran source for XERBLA with +noppu and provide both xerbla and xerbla_ entry points:

```
SUBROUTINE XERBLA(SRNAME,INFO)
ENTRY XERBLA_SRNAME,INFO)
...
```

- Use f90 ALIAS directives and provide both xerbla and xerbla_ entry points:

```
!$HP$ ALIAS xerbla='xerbla'
!$HP$ ALIAS xerbla_='xerbla_'
SUBROUTINE XERBLA(SRNAME,INFO)
ENTRY XERBLA_(SRNAME,INFO)
...
```

  The ALIAS directives prevent the compiler from postpending the underbar onto the entry points and external references to XERBLA.

Use the following workaround on Itanium if you are compiling your own C version of XERBLA:

- Undefine any previous aliasing from include files and provide both xerbla and xerbla_ entry points explicitly:

```
#undef xerbla
#undef xerbla_

void xerbla (char*name, int*iarg, size_t len_name){
...
}

void xerbla_ (char*name, int*iarg, size_t len_name){
xerbla (name, iarg, len_name);
}
```

  This problem is not confined to the XERBLA subroutine. It occurs on Itanium when the following conditions are met:

  - Any subprogram with the same name as a user-visible MLIB subprogram is compiled in your program with
  - `+noppu`
  - Another MLIB subprogram used by your program also calls that subprogram

PA-RISC and Itanium processors

Also note that all the workarounds listed for Itanium are more generic and can also be used for PA-RISC. Therefore, if you are coding your own version of a MLIB routine called "mlib_routine" on PA-RISC or Itanium, a Fortran version might be implemented as:

```
!$HP$ ALIAS mlib_routine='mlib_routine'
!$HP$ ALIAS mlib_routine_='mlib_routine_'
```

```
SUBROUTINE mlib_routine(...)
ENTRY   mlib_routine_(...)
...
```

And a C version might be:

```
#undef mlib_routine
#undif mlib_routine_

void mlib_routine (...){
...
}

void mlib_routine_(...){
mlib_routine(...);
}
```

Optimization

The key computational kernels in VECLIB have been optimized to take full advantage of both PA-RISC and Itanium tightly integrated architectures. 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 3 for instructions on linking the desired library with your program.

Parallel processing

Parallel processing is available on multi-processor HP platforms running the HP-UX 11i or greater operating system and Linux. 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 runtime. 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. On HP-UX systems, use the **+O3** and **+Oparallel** compiler options to get a multithreaded executable when you link with the HP Fortran compiler; use **+O3** and **+Oopenmp** when you link with the HP C compiler; and use **+Oopenmp** when you link with the HP aC++ compiler:

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

cc [*options* including **+O3 +Oopenmp**] *file* ... **–Wl,–aarchive_shared –lveclib –lcl –lm**

aCC [*options* including **+Oopenmp**] *file* ... **–Wl,–aarchive_shared –lveclib –lcl –lm**

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

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

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

aCC [*options*] *file* ... **–Wl,–aarchive_shared –lveclib –lcl –lm**

VECLIB for Linux is always multithreaded enabled.

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 the subroutine 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.

- To modify the effect of the MLIB_NUMBER_OF_THREADS and MLIB_SETNUMTHREADS, you can also use MP_NUMBER_OF_THREADS at runtime. When MP_NUMBER_OF_THREADS is set to a value smaller than MLIB_NUMBER_OF_THREADS and MLIB_SETNUMTHREADS, the value of MP_NUMBER_OF_THREADS takes precedence and limits the amount of parallelism to MP_NUMBER_OF_THREADS.

These controls set the maximum amount of SMP 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 runtime if multiple processors are available. If multiple processors are available, VECLIB detects whether the program is already using multiple threads. If the application calling VECLIB is not running multiple threads, VECLIB will attempt to use all available threads (limited by MLIB_NUMBER_THREADS). If the application calling VECLIB is already running multiple threads, VECLIB will attempt to use the remaining threads without over-subscribing. This is a form of nested parallelism.

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. VECLIB supports nested parallelism where the outer parallelism is implemented through OpenMP while the inner parallelism is implemented with VECLIB SMP parallelism. 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 defining OpenMP parallel regions.

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

```
      NTHREADS = 4
C$OMP PARALLEL DO NUM_THREADS(NTHREADS)
      DO J=1, N
         CALL DAXPY (N-I,A(I,J),A(I+1,I),1,A(I+1,J),1)
      ENDO
C$OMP END PARALLEL DO
```

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. OpenMP supports the `PARALLEL` and `END PARALLEL` directions that define a block of code that is to be executed by multiple threads in parallel.

OpenMP-based nested parallelism

Nested parallelism can be achieved when calling VECLIB parallelized subprograms from an OpenMP parallel region. (See "Parallelized subprograms in VECLIB" on page 1104.) Consider the following code running on an HP platform with at least four processors:

```
      ...
      call omp_set_nested (.true.)
c$omp parallel NUM_THREADS(2)
      myid = omp_get_thread_num
      if (myid.eq.0) then
        call dgemm('n', 'n', m, m, m, alpha, a, lda, b, ldb, beta,
          c,ldc)
      else
        call dgemm('n', 'n', m, m, m, alpha, d, ldd, e, lde, beta,
          f,ldf)
```

```
      endif
c$omp end parallel
      call omp_set_nested(.false.)
      ...
```

Using MLIB_NUMBER_OF_THREADS set to 1, the code would run two-way parallel: one OpenMP thread for

$$C = \alpha AB + \beta C$$

and another for

$$F = \alpha DE + \beta F$$

Setting MLIB_NUMBER_OF_THREADS to 2 would allow nested parallelism and run the code four-way parallel.

If a parallel VECLIB subprogram is called from a parallelized loop or region, VECLIB will automatically avoid over-subscription of the CPUs. The number of threads spawned by each call to a parallelized VECLIB subroutine on a nested parallel region is limited by:

- MLIB_NUMBER_OF_THREADS
- The number of threads still available in the system
- will never be larger than four. Specifically:

  `MIN (MLIB_NUMBER_OF_THREADS, threads still available, 4)`

Message passing-based nested parallelism

Nested parallelism can be achieved when calling VECLIB parallelized subprograms from an MPI process. (See “Parallelized subprograms in VECLIB” on page 1104.) Consider the following code:

```
...
call mpi_init (ierr)
call mpi_comm_rank(MPI_COMM_WORLD, myid, ierr)
if (myid.eq.0) then
   call dgemm('n', 'n', m, m, m, alpha, a, lda, b, ldb, beta,
     c,ldc)
else
  call dgemm('n', 'n', m, m, m, alpha, d, ldd, e, lde, beta,
     f,ldf)
endif
...
```

Assume the application started on two MPI processes. Using MLIB_NUMBER_OF_THREADS set to 1, the code would run two-way parallel: one MPI process for

$$C = \alpha AB + \beta C$$

and another for

$$F = \alpha DE + \beta F$$

Setting MLIB_NUMBER_OF_THREADS to 2 would allow nested parallelism and run the code four-way parallel.

Default CPS library stack is too small for MLIB

In `libcps`, the HP Compiler Parallel Support library, a CPS thread has a default stack size of 8M bytes. For performance reasons, several subprograms in HP MLIB use the stack for temporary arrays that exceed the default value. Using the default CPS stack size, these routines overwrite neighboring stacks, resulting in errors that are difficult to diagnose.

The solution is to change the CPS thread stacksize attribute to a value that is large enough to accommodate all the MLIB subprograms the thread may encounter. Currently, 8 MB*(the number of threads) should be sufficient for all MLIB subprograms.

The environment variable CPS_STACK_SIZE expects values in K bytes. Setting the stack size as follows would be sufficient for programs that execute on two threads:

For C shell:

```
% setenv CPS_STACK_SIZE 16384
```

For Korn shell:

```
% export CPS_STACK_SIZE=16384
```

Default Pthread library stack is too small for MLIB

The stack allocated for each new thread created using direct pthread calls to “pthread_create” might not be large enough for HP MLIB. Several subprograms in HP MLIB use the stack for storing temporary work arrays and improve performance. If the stack size is not large enough, these routines overwrite neighboring stacks, resulting in errors that are difficult to diagnose.

Currently, 8 MB*(the number of threads) should be sufficient for all MLIB subprograms. If your application launchs threads directly from pthread library calls, set the minimum allocated stack size to match HP MLIB needs on each new thread. Setting the stack size on HP-UX as follows would be sufficient for programs that execute on two threads:

```
int stacksize = 8388608;
(...)
pthread_attr_setstacksize(&thread_attr, stacksize);
pthread_create(&thread_id, &thread_attr, thread_func, &thread_parm);
(...)
```

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.

Subprograms in VECLIB are provided in both 32-bit and 64-bit addressing versions, except in the VECLIB8 library where only 64-bit addressing is supported.

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 1-3:

Table 1-3 VECLIB Naming Convention—Data Type

T	Data Type	VECLIB	VECLIB8
S	Single Precision	REAL*4	REAL*4
D	Double Precision	REAL*8	REAL*8
I	Integer	INTEGER*4	INTEGER*8
C	Complex	COMPLEX*8	COMPLEX*8
Z	Double Complex	COMPLEX*16	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.

Fortran 77 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 F_ZSUM. Refer to “Legacy BLAS routines” on page 211 and “BLAS Standard routines” on page 339 for more information.

C BLAS Standard subprograms begin with *C_* prepended to each routine name.

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 1-3), and the byte lengths of its arguments. This relationship is shown in Table :

MLIB Argument Lengths

T	**MLIB Argument Lengths**			
	VECLIB INTEGER LOGICAL	VECLIB8 INTEGER LOGICAL	**REAL**	**COMPLEX**
S	4	8	4	8
D	4	8	8	16
C	4	8	4	8
Z	4	8	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 1-4 on page 27 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 Fortran 77.

Table 1-4 lists the operator arguments and their associated named constants.

Table 1-4 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 337, “XERVEC” on page 623, and “F_BLASERROR” on page 605 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 8.5 supports a subset of the BLAS Standard routines. This chapter describes dense and banded BLAS Standard functionality in “BLAS Standard routines” on page 152.

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, $i = 1, 2, \ldots, N$, is stored in X. If $INCX \geq 0$, 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 25 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 152 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 152 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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       i, IIAMAX, n, incx, x(lenx)
i = IIAMAX(n, x, incx)

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       i, IIAMIN, n, incx, x(lenx)
i = IIAMIN(n, x, incx)

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       i, IICTxx, n, incx, a, x(lenx)
i = IICTxx(n, x, incx, a)

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

INTEGER*8       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 **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, 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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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

VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       i, IISVxx, n, incx, a, x(lenx)
i = IISVxx(n, x, incx, a)

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

INTEGER*8       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 $\mathbf{x}((i-1)\times|\mathbf{incx}|+1)$.

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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, s, IAMAX, x(lenx)
s = IAMAX(n, x, incx)

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, s, IAMIN, x(lenx)
s = IAMIN(n, x, incx)

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, s, IASUM, x(lenx)
s = IASUM(n, x, incx)

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

INTEGER*8       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

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

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 VECLIB:

```
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)
```

VECLIB8:

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

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

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

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

INTEGER*8       n, incx, incy
COMPLEX*16      a, x(lenx), y(leny)
CALL ZAXPY(n, a, x, incx, y, incy)

INTEGER*8       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

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

Usage

VECLIB:

```
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)
```

VECLIB8:

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

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

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

INTEGER*8        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 **m** ≤ 0 or **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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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 $\mathbf{n} \le 0$, 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)×**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 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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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)×**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 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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, incy, x(lenx), y(leny)
CALL ICOPY(n, x, incx, y, incy)

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

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

INTEGER*8       n, incx, incy
COMPLEX*16      x(lenx), y(leny)
CALL ZCOPY(n, x, incx, y, incy)

INTEGER*8       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 83.

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 VECLIB:

```
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)
```

VECLIB8:

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

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

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

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

INTEGER*8 n, incx, incy
COMPLEX*16 s, ZDOTC, x(lenx), y(leny)
s = ZDOTC(n, x, incx, y, incy)

INTEGER*8 n, incx, incy
COMPLEX*16 s, ZDOTU, x(lenx), y(leny)
s = ZDOTU(n, x, incx, y, incy)

INTELCOMP:

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

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

INTEGER*4 n, incx, incy
INTEGER*8 n, incx, incy
COMPLEX*16 s, ZDOTC, x(lenx), y(leny), res
s = ZDOTC(n, x, incx, y, incy, res)

INTEGER*4 n, incx, incy
INTEGER*8 n, incx, incy
COMPLEX*16 s, ZDOTU, x(lenx), y(leny), res
s = ZDOTU(n, x, incx, y, incy, res)

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.
	res	Pointer to output (for use with Intel compilers only).
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 $\{k_1, k_2, ..., k_m\}$ 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 VECLIB:

```
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)
```

VECLIB8:

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

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

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

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

INTEGER*8       m, indx(m)
COMPLEX*16      s, ZDOTCI, x(m), y(n)
s = ZDOTCI(m, x, indx, y)

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

INTEGER*8       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** ≠ 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

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

Usage

VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       m, indx(m), y(n), x(m)
CALL IGTHR(m, y, x, indx)
```

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

INTEGER*8       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, $\mathbf{y}(i) = y_i$. 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

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

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 $\mathbf{indx}(j) = i$.

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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8          m, indx(m), y(n), x(m)
CALL IGTHRZ(m, y, x, indx)

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

INTEGER*8          m, indx(m)
COMPLEX*16         y(n), x(m)
CALL ZGTHRZ(m, y, x, indx)
```

Input

m Number of interesting elements of y, $\mathbf{m} \leq \mathbf{n}$, where **n** is the length of **y**. If $\mathbf{m} \leq 0$, 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 $\mathbf{m} \leq 0$, 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

VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, nindx, indx(n), a, x(lenx)
CALL ILSTxx(n, x, incx, a, nindx, indx)

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

INTEGER*8       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 $\mathbf{x}((i-1)\times|\mathbf{incx}|+1)$.

Notes These subprograms are sometimes useful for optimizing a loop containing an **IF** statement. Refer to “Example 2” on page 101.

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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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** = −∞, 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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8       n, incx
REAL*4          s, SNRM2, x(lenx)
s = SNRM2(n, x, incx)

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

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

INTEGER*8 **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 VECLIB:

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)

VECLIB8:

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

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

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

```
INTEGER*8       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 **n** ≤ 0, then **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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       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)×|**incx**|+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 π/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 VECLIB:

```
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)
```

VECLIB8:

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

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

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

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

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

INTEGER*8       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 126 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 VECLIB, VECLIB8:

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 VECLIB:

```
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)
```

VECLIB8:

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

INTEGER*8        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

VECLIB:

```
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)
```

VECLIB8:

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

INTEGER*8       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**$((i-1)\times$**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.

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 117 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 VECLIB, VECLIB8:

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 VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8       n, incx
REAL*4          a, x(lenx)
CALL SRSCL(n, a, x, incx)
```

```
INTEGER*8       n, incx
REAL*8          a, x(lenx)
CALL DRSCL(n, a, x, incx)

INTEGER*8       n, incx
COMPLEX*8       a, x(lenx)
CALL CRSCL(n, a, x, incx)

INTEGER*8       n, incx
REAL*4          a
COMPLEX*8       x(lenx)
CALL CSRSCL(n, a, x, incx)

INTEGER*8       n, incx
COMPLEX*16      a, x(lenx)
CALL ZRSCL(n, a, x, incx)

INTEGER*8       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** $\neq 0$.

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 |**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

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 VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8       n, incx
REAL*4          a, x(lenx)
CALL SSCAL(n, a, x, incx)

INTEGER*8       n, incx
REAL*8          a, x(lenx)
CALL DSCAL(n, a, x, incx)

INTEGER*8       n, incx
COMPLEX*8       a, x(lenx)
CALL CSCAL(n, a, x, incx)

INTEGER*8       n, incx
REAL*4          a
COMPLEX*8       x(lenx)
CALL CSSCAL(n, a, x, incx)

INTEGER*8       n, incx
COMPLEX*8       a, x(lenx)
CALL CSCALC(n, a, x, incx)

INTEGER*8       n, incx
COMPLEX*16      a, x(lenx)
CALL ZSCAL(n, a, x, incx)

INTEGER*8       n, incx
REAL*8          a
COMPLEX*16      x(lenx)
CALL ZDSCAL(n, a, x, incx)

INTEGER*8       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 **n** ≤ 0, the subprograms do not reference **x**.
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 CSCALC

and ZSCALC and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array **x**, **incx** ≠ 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 **n** ≤ 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

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

Usage VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       m, indx(m), x(m), y(n)
CALL ISCTR(m, x, indx, y)
```

```
INTEGER*8       m, indx(m)
COMPLEX*8       x(m), y(n)
CALL CSCTR(m, x, indx, y)

INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, s, ISUM, x(lenx)
s = ISUM(n, x, incx)
```

```
INTEGER*8       n, incx
COMPLEX*8       s, CSUM, x(lenx)
s = CSUM(n, x, incx)

INTEGER*8       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 **n** ≤ 0, then **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 VECLIB:

```
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)
```

VECLIB8:

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

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

INTEGER*8       n, incx, incy, x(lenx), y(leny)
CALL ISWAP(n, x, incx, y, incy)
```

```
INTEGER*8       n, incx, incy
COMPLEX*8       x(lenx), y(leny)
CALL CSWAP(n, x, incx, y, incy)

INTEGER*8       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

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

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 VECLIB:

```
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)
```

VECLIB8:

INTEGER*8 n, incw, incx, incy
REAL*4 s, SWDOT, w(lenw), x(lenx), y(leny)
s = SWDOT(n, w, incw, x, incx, y, incy)

INTEGER*8 n, incw, incx, incy
REAL*8 s, DWDOT, w(lenw), x(lenx), y(leny)
s = DWDOT(n, w, incw, x, incx, y, incy)

INTEGER*8 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*8 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*8 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*8 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)×**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 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 VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8       n, incx
REAL*4          x(lenx)
CALL SZERO(n, x, incx)

INTEGER*8       n, incx
REAL*8          x(lenx)
CALL DZERO(n, x, incx)

INTEGER*8       n, incx, x(lenx)
CALL IZERO(n, x, incx)

INTEGER*8       n, incx
COMPLEX*8       x(lenx)
CALL CZERO(n, x, incx)
```

```
INTEGER*8       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** ≠ 0. 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

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 VECLIB:

```
INTEGER*4       INCX, K, N
REAL*4          R, X( * )
SUBROUTINE F_SAMAX_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*8          R, X( * )
SUBROUTINE F_DAMAX_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMAX_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*8          R
COMPLEX*16      X( * )
SUBROUTINE F_ZAMAX_VAL (N, X, INCX, K, R)
```

VECLIB8:

```
INTEGER*8       INCX, K, N
REAL*4          R, X( * )
SUBROUTINE F_SAMAX_VAL (N, X, INCX, K, R)

INTEGER*8       INCX, K, N
REAL*8          R, X( * )
SUBROUTINE F_DAMAX_VAL (N, X, INCX, K, R)

INTEGER*8       INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMAX_VAL (N, X, INCX, K, R)
```

```
INTEGER*8       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

$$x_k = \max(|Re(x_i)| + |Im(x_i)|) \qquad (\text{where } 0 \le i < n)$$

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 VECLIB:

```
INTEGER*4       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SAMIN_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DAMIN_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMIN_VAL (N, X, INCX, K)

INTEGER*4       INCX, K, N
REAL*8          R
COMPLEX*16      X( * )
SUBROUTINE F_ZAMIN_VAL (N, X, INCX, K)
```

VECLIB8:

```
INTEGER*8       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SAMIN_VAL (N, X, INCX, K, R)

INTEGER*8       INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DAMIN_VAL (N, X, INCX, K, R)
```

```
INTEGER*8       INCX, K, N
REAL*4          R
COMPLEX*8       X( * )
SUBROUTINE F_CAMIN_VAL (N, X, INCX, K)

INTEGER*8       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,..., *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 VECLIB:

```
INTEGER*4       INCX, INCY, N
REAL*4          C, S, X( * ), Y( * )
SUBROUTINE F_SAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
REAL*8          C, S, X( * ), Y( * )
SUBROUTINE F_DAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
REAL*4          C
COMPLEX*8       S, X( * ), Y( * )
SUBROUTINE F_CAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
REAL*8          C
COMPLEX*16      S, X( * ), Y( * )
SUBROUTINE F_ZAPPLY_GROT (N, C, S, X, INCX, Y, INCY)
```

VECLIB8:

```
INTEGER*8       INCX, INCY, N
REAL*4          C, S, X( * ), Y( * )
SUBROUTINE F_SAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER*8       INCX, INCY, N
REAL*8          C, S, X( * ), Y( * )
SUBROUTINE F_DAPPLY_GROT (N, C, S, X, INCX, Y, INCY)
```

```
INTEGER*8       INCX, INCY, N
REAL*4          C
COMPLEX*8       S, X( * ), Y( * )
SUBROUTINE F_CAPPLY_GROT (N, C, S, X, INCX, Y, INCY)

INTEGER*8       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 VECLIB:

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

INTEGER*4       INCX, INCY, N
REAL*8          ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_DAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER*4       INCX, INCY, N
COMPLEX*8       ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_CAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER*4       INCX, INCY, N
COMPLEX*16      ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_ZAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)
```

VECLIB8:

```
INTEGER*8       INCX, INCY, N
REAL*4          ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_SAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER*8       INCX, INCY, N
REAL*8          ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_DAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER*8       INCX, INCY, N
COMPLEX*8       ALPHA, BETA, X( * ), Y( * )
SUBROUTINE F_CAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCW, INCV, INCU, N
REAL*4          ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_SAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER*4       INCW, INCV, INCU, N
REAL*8          ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_DAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER*4       INCW, INCV, INCU, N
COMPLEX*8       ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_CAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER*4       INCW, INCV, INCU, N
COMPLEX*16      ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_ZAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )
```

VECLIB8:

```
INTEGER*8       INCW, INCV, INCU, N
REAL*4          ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_SAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER*8       INCW, INCV, INCU, N
REAL*8          ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_DAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )

INTEGER*8       INCW, INCV, INCU, N
COMPLEX*8       ALPHA, R, W( * ), V( * ), U( * )
SUBROUTINE F_CAXPY_DOT ( N, ALPHA, W, INCW, V, INCV, U, INCU )
```

```
INTEGER*8       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 VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8        INCX, INCY, N
REAL*4           X( * ), Y( * )
SUBROUTINE F_SCOPY (N, X, INCX, Y, INCY)

INTEGER*8        INCX, INCY, N
REAL*8           X( * ), Y( * )
SUBROUTINE F_DCOPY (N, X, INCX, Y, INCY)

INTEGER*8        INCX, INCY, N
COMPLEX*8        X( * ), Y( * )
SUBROUTINE F_CCOPY (N, X, INCX, Y, INCY)

INTEGER*8        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 VECLIB:

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

INTEGER*4       CONJ, INCX, INCY, N
REAL*8          ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_DDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER*4       CONJ, INCX, INCY, N
COMPLEX*8       ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_CDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER*4       CONJ, INCX, INCY, N
COMPLEX*16      ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_ZDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)
```

VECLIB8:

```
INTEGER*8       CONJ, INCX, INCY, N
REAL*4          ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_SDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)
```

```
INTEGER*8       CONJ, INCX, INCY, N
REAL*8          ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_DDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER*8       CONJ, INCX, INCY, N
COMPLEX*8       ALPHA, BETA, R, X( * ), Y( * )
SUBROUTINE F_CDOT (CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R)

INTEGER*8       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 VECLIB:

```
INTEGER*4       CMACH
REAL*4          FUNCTION F_SFPINFO (CMACH)

INTEGER*4       CMACH
REAL*4          FUNCTION F_DFPINFO (CMACH)
```

VECLIB8:

```
INTEGER*8       CMACH
REAL*4          FUNCTION F_SFPINFO (CMACH)

INTEGER*8       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 $1/\text{SFMIN}$ does not overflow. If $1/\text{OV} < \text{UN}$, 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 VECLIB, VECLIB8

```
REAL*4          C
REAL*4          A, B, R, S
SUBROUTINE F_SGEN_GROT (A, B, C, S, R)

REAL*8          C
REAL*8          A, B, R, S
SUBROUTINE F_DGEN_GROT (A, B, C, S, R)

REAL*4          C
COMPLEX*8       A, B, R, S
SUBROUTINE F_CGEN_GROT (A, B, C, S, R)

REAL*8          C
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 v. If n is less than or equal to zero, this function returns immediately with the output scalar **tau** set to zero.

Usage VECLIB:

```
INTEGER*4       INCX, N
REAL*4          ALPHA, TAU
REAL*4          X ( * )
SUBROUTINE F_SGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*4       INCX, N
REAL*8          ALPHA, TAU
REAL*8          X ( * )
SUBROUTINE F_DGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*4       INCX, N
COMPLEX*8       ALPHA, TAU
COMPLEX*8       X ( * )
SUBROUTINE F_CGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*4       INCX, N
COMPLEX*16      ALPHA, TAU
COMPLEX*8       X ( * )
SUBROUTINE F_ZGEN_HOUSE ( N, ALPHA, X, INCX, TAU )
```

VECLIB8:

```
INTEGER*8       INCX, N
REAL*4          ALPHA, TAU
REAL*4          X ( * )
SUBROUTINE F_SGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*8       INCX, N
REAL*8          ALPHA, TAU
REAL*8          X ( * )
SUBROUTINE F_DGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*8       INCX, N
COMPLEX*8       ALPHA, TAU
COMPLEX*8       X ( * )
SUBROUTINE F_CGEN_HOUSE ( N, ALPHA, X, INCX, TAU )

INTEGER*8       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 VECLIB:

INTEGER*4 JROT
REAL*4 S, X, Y, Z, C
SUBROUTINE F_SGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*4 JROT
REAL*8 S, X, Y, Z, C
SUBROUTINE F_DGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*4 JROT
REAL*4 X, Z, C
COMPLEX*8 S, Y
SUBROUTINE F_CGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*4 JROT
REAL*8 X, Z, C
COMPLEX*16 S, Y
SUBROUTINE F_ZGEN_JROT (JROT, X, Y, Z, C, S)

VECLIB8:

```
INTEGER*8       JROT
REAL*4          S, X, Y, Z, C
SUBROUTINE F_SGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*8       JROT
REAL*8          S, X, Y, Z, C
SUBROUTINE F_DGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*8       JROT
REAL*4          X, Z, C
COMPLEX*8       S, Y
SUBROUTINE F_CGEN_JROT (JROT, X, Y, Z, C, S)

INTEGER*8       JROT
REAL*8          X, Z, C
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 VECLIB:

```
INTEGER*4       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SMAX_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DMAX_VAL (N, X, INCX, K, R)
```

VECLIB8:

```
INTEGER*8       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SMAX_VAL (N, X, INCX, K, R)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SMIN_VAL (N, X, INCX, K, R)

INTEGER*4       INCX, K, N
REAL*8          R
REAL*8          X( * )
SUBROUTINE F_DMIN_VAL (N, X, INCX, K, R)
```

VECLIB8:

```
INTEGER*8       INCX, K, N
REAL*4          R
REAL*4          X( * )
SUBROUTINE F_SMIN_VAL (N, X, INCX, K, R)

INTEGER*8       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 VECLIB:

```
INTEGER*4     INCX, N, NORM
REAL*4        X( * )
REAL*4        FUNCTION F_SNORM (NORM, N, X, INCX)

INTEGER*4     INCX, N, NORM
REAL*8        X( * )
REAL*4        FUNCTION F_SNORM (NORM, N, X, INCX)
```

VECLIB8:

```
INTEGER*8     INCX, N, NORM
REAL*4        X( * )
REAL*4        FUNCTION F_SNORM (NORM, N, X, INCX)

INTEGER*8     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, $P = E_n \ldots E_1$ 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 $incp > 0$, $x(i) \leftrightarrow x(p(i))$

Usage VECLIB:

```
INTEGER*4       INCP, INCX, N
INTEGER         P( * )
REAL*4          X( * )
SUBROUTINE F_SPERMUTE (N, P, INCP, X, INCX)

INTEGER*4       INCP, INCX, N
INTEGER         P( * )
REAL*8          X( * )
SUBROUTINE F_DPERMUTE (N, P, INCP, X, INCX)

INTEGER*4       INCP, INCX, N
INTEGER         P( * )
COMPLEX*8       X( * )
SUBROUTINE F_CPERMUTE (N, P, INCP, X, INCX)

INTEGER*4       INCP, INCX, N
INTEGER         P( * )
COMPLEX*16      X( * )
SUBROUTINE F_ZPERMUTE (N, P, INCP, X, INCX)
```

VECLIB8:

```
INTEGER*8       INCP, INCX, N
INTEGER         P( * )
REAL*4          X( * )
SUBROUTINE F_SPERMUTE (N, P, INCP, X, INCX)
```

```
INTEGER*8       INCP, INCX, N
INTEGER         P( * )
REAL*8          X( * )
SUBROUTINE F_DPERMUTE (N, P, INCP, X, INCX)

INTEGER*8       INCP, INCX, N
INTEGER         P( * )
COMPLEX*8       X( * )
SUBROUTINE F_CPERMUTE (N, P, INCP, X, INCX)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, N
REAL*4          ALPHA, X( * )
SUBROUTINE F_SRSCALE (N, ALPHA, X, INCX)

INTEGER*4       INCX, N
REAL*8          ALPHA, X( * )
SUBROUTINE F_DRSCALE (N, ALPHA, X, INCX)

INTEGER*4       INCX, N
REAL*4          ALPHA
COMPLEX*8       X( * )
SUBROUTINE F_CRSCALE (N, ALPHA, X, INCX)

INTEGER*4       INCX, N
REAL*4          ALPHA
COMPLEX*8       X( * )
SUBROUTINE F_ZRSCALE (N, ALPHA, X, INCX)
```

VECLIB8:

```
INTEGER*8       INCX, N
REAL*4          ALPHA, X( * )
SUBROUTINE F_SRSCALE (N, ALPHA, X, INCX)

INTEGER*8       INCX, N
REAL*8          ALPHA, X( * )
SUBROUTINE F_DRSCALE (N, ALPHA, X, INCX)

INTEGER*8       INCX, N
REAL*4          ALPHA
COMPLEX*8       X( * )
SUBROUTINE F_CRSCALE (N, ALPHA, X, INCX)
```

```
INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, N, SORT
REAL*4          X( * )
SUBROUTINE F_SSORT (SORT, N, X, INCX)

INTEGER*4       INCX, N, SORT
REAL*8          X( * )
SUBROUTINE F_DSORT (SORT, N, X, INCX)
```

VECLIB8:

```
INTEGER*8       INCX, N, SORT
REAL*4          X( * )
SUBROUTINE F_SSORT (SORT, N, X, INCX)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCP, INCX, N, SORT
INTEGER         P( * )
REAL*4          X( * )
SUBROUTINE F_SSORTV (SORT, N, X, INCX, P, INCP)

INTEGER*4       INCP, INCX, N, SORT
INTEGER         P( * )
REAL*8          X( * )
SUBROUTINE F_DSORTV (SORT, N, X, INCX, P, INCP)
```

VECLIB8:

```
INTEGER*8       INCP, INCX, N, SORT
INTEGER         P( * )
REAL*4          X( * )
SUBROUTINE F_SSORTV (SORT, N, X, INCX, P, INCP)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, N
REAL*4          X( * )
REAL*4          FUNCTION F_SSUM (N, X, INCX)

INTEGER*4       INCX, N
REAL*8          X( * )
REAL*8          FUNCTION F_DSUM (N, X, INCX)

INTEGER*4       INCX, N
COMPLEX*8       X( * )
COMPLEX*8       FUNCTION F_CSUM (N, X, INCX)

INTEGER*4       INCX, N
COMPLEX*16      X( * )
COMPLEX*16      FUNCTION F_ZSUM (N, X, INCX)
```

VECLIB8:

```
INTEGER*8       INCX, N
REAL*4          X( * )
REAL*4          FUNCTION F_SSUM (N, X, INCX)

INTEGER*8       INCX, N
REAL*8          X( * )
REAL*8          FUNCTION F_DSUM (N, X, INCX)

INTEGER*8       INCX, N
COMPLEX*8       X( * )
COMPLEX*8       FUNCTION F_CSUM (N, X, INCX)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, N
REAL*4          SCL, SSQ, X( * )
SUBROUTINE F_SSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER*4       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)
```

VECLIB8:

```
INTEGER*8       INCX, N
REAL*4          SCL, SSQ, X( * )
SUBROUTINE F_SSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER*8       INCX, N
REAL*8          SCL, SSQ, X( * )
SUBROUTINE F_DSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER*8       INCX, N
REAL*4          SCL, SSQ
COMPLEX*8       X( * )
SUBROUTINE F_CSUMSQ (N, X, INCX, SSQ, SCL)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCX, INCY, N
REAL*4          X( * ), Y( * )
SUBROUTINE F_SSWAP (N, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
REAL*8          X( * ), Y( * )
SUBROUTINE F_DSWAP (N, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
COMPLEX*8       X( * ), Y( * )
SUBROUTINE F_CSWAP (N, X, INCX, Y, INCY)

INTEGER*4       INCX, INCY, N
COMPLEX*16      X( * ), Y( * )
SUBROUTINE F_ZSWAP (N, X, INCX, Y, INCY)
```

VECLIB8:

```
INTEGER*8       INCX, INCY, N
REAL*4          X( * ), Y( * )
SUBROUTINE F_SSWAP (N, X, INCX, Y, INCY)

INTEGER*8       INCX, INCY, N
REAL*8          X( * ), Y( * )
SUBROUTINE F_DSWAP (N, X, INCX, Y, INCY)

INTEGER*8       INCX, INCY, N
COMPLEX*8       X( * ), Y( * )
SUBROUTINE F_CSWAP (N, X, INCX, Y, INCY)

INTEGER*8       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 VECLIB:

```
INTEGER*4       INCW, INCX, INCY, N
REAL*4          ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_SWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER*4       INCW, INCX, INCY, N
REAL*8          ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_DWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER*4       INCW, INCX, INCY, N
COMPLEX*8       ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_CWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER*4       INCW, INCX, INCY, N
COMPLEX*16      ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_ZWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)
```

VECLIB8:

```
INTEGER*8       INCW, INCX, INCY, N
REAL*4          ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_SWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER*8       INCW, INCX, INCY, N
REAL*8          ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_DWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)
```

```
INTEGER*8       INCW, INCX, INCY, N
COMPLEX*8       ALPHA, BETA, W( * ), X( * ), Y( * )
SUBROUTINE F_CWAXPBY (N, ALPHA, X, INCX, BETA, Y, INCY, W,
INCW)

INTEGER*8       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. Refer to “BLAS Standard routines” on page 339.

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 211 and “BLAS Standard routines” on page 339.

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 25 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 339 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 339 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 355 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 VECLIB:

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)

VECLIB8:

CHARACTER*1 trans
INTEGER*8 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*8 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*8 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*8 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, $0 \le$ **kl** $<$ **n**.
ku	The upper bandwidth of A, that is, the number of nonzero diagonals above the principal diagonal in the band, $0 \le$ **ku** $<$ **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** $\ge$ **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 358 for a description of the BLAS Standard subprograms for general matrix copy.

Usage VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     trans
INTEGER*8       m, n, lda, ldb
REAL*4          a(lda, *), b(ldb, *)
CALL SGECPY(trans, m, n, a, lda, b, ldb)

CHARACTER*1     trans
INTEGER*8       m, n, lda, ldb
REAL*8          a(lda, *), b(ldb, *)
CALL DGECPY(trans, m, n, a, lda, b, ldb)
```

```
CHARACTER*1     trans
INTEGER*8       m, n, lda, ldb
COMPLEX*8       a(lda, *), b(ldb, *)
CALL CGECPY(trans, m, n, a, lda, b, ldb)

CHARACTER*1     trans
INTEGER*8       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, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a** or **b**.

n Number of columns in matrix A, **n** ≥ 0. If **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 362 for a description of the BLAS Standard subprograms for general matrix-matrix multiply.

Usage VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     transa, transb
INTEGER*8       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*8       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*8       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*8       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, **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

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 and CGEMM and SGEMM in VECLIB8. 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 and VECLIB8 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:

$$C \leftarrow \alpha AB + \beta C, \quad C \leftarrow \alpha A^T B + \beta C, \quad C \leftarrow \alpha A^* B + \beta C,$$
$$C \leftarrow \alpha AB^T + \beta C, \quad C \leftarrow \alpha A^T B^T + \beta C, \quad C \leftarrow \alpha A^* B^T + \beta C,$$
$$C \leftarrow \alpha AB^* + \beta C, \quad C \leftarrow \alpha A^T B^* + \beta C, \quad C \leftarrow \alpha A^* B^* + \beta C.$$

Refer to "F_SGEMM/F_DGEMM/F_CGEMM/F_ZGEMM" on page 362 for a description of the equivalent BLAS Standard subprograms.

Usage

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     transa, transb
INTEGER*8       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*8       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** ≥ 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

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:

$$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ \text{and}\ y \leftarrow \alpha A^* x + \beta y.$$

Refer to "F_SGEMV/F_DGEMV/F_CGEMV/F_ZGEMV" on page 365 for a description of the BLAS Standard subprograms for a triangular matrix-vector multiply.

Usage VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     trans
INTEGER*8       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*8       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*8       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*8       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

$$A \leftarrow axy^T + A \qquad \text{and} \qquad A \leftarrow axy^* + A,$$

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 375 for a description of the BLAS Standard subprograms for general rank-1 update.

Usage VECLIB:

```
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)
```

VECLIB8:

```
INTEGER*8       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*8       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*8       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*8       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*8       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*8       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**($(i-1)\times$**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 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 SGETRA/DGETRA/CGETRA/ZGETRA
In-place transpose of a general square matrix

Purpose These subprograms overwrite an n by n matrix A with its transpose.

Usage VECLIB:

```
INTEGER*4       n, lda
REAL*4          a(lda, n)
CALL SGETRA(n, a, lda)

INTEGER*4       n, lda
REAL*8          a(lda, n)
CALL DGETRA(n, a, lda)

CHARACTER*1     trans
INTEGER*4       n, lda
COMPLEX*8       a(lda, n)
CALL CGETRA(trans, n, a, lda)

CHARACTER*1     trans
INTEGER*4       n, lda
COMPLEX*16      a(lda, n)
CALL ZGETRA(trans, n, a, lda)
```

VECLIB8:

```
INTEGER*8       n, lda
REAL*4          a(lda, n)
CALL SGETRA(n, a, lda)

INTEGER*8       n, lda
REAL*8          a(lda, n)
CALL DGETRA(n, a, lda)

CHARACTER*1     trans
INTEGER*8       n, lda
COMPLEX*8       a(lda, n)
CALL CGETRA(trans, n, a, lda)

CHARACTER*1     trans
INTEGER*8       n, lda
COMPLEX*16      a(lda, n)
CALL ZGETRA(trans, n, a, lda)
```

Input **trans** Transposition option for A:

		trans = 'T' or 't' Compute A <- A-transpose trans = 'C' or 'c' Compute A <- A conjugate-transpose
	n	Number of rows and columns in matrix *A*, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**.
	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).
Output	**a**	The result 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 (refer to the end of this chapter) can be replaced with a user-supplied version to change the error procedure. Error conditions are

trans != 'T' or 't' or 'C' or 'c'
n < 0, and
lda < max(**n**,1)

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 trans argument as 'Transposed' for 'T', or 'Conjugate-Transposed' for 'C'.

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 378 and "F_CHBMV/F_ZHBMV" on page 340 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 $kd = 2$:

$$\begin{bmatrix} 11 & 12 & 13 & 0 & 0 & 0 & 0 \\ 12 & 22 & 23 & 24 & 0 & 0 & 0 \\ 13 & 23 & 33 & 34 & 35 & 0 & 0 \\ 0 & 24 & 34 & 44 & 45 & 46 & 0 \\ 0 & 0 & 35 & 45 & 55 & 56 & 57 \\ 0 & 0 & 0 & 46 & 56 & 66 & 67 \\ 0 & 0 & 0 & 0 & 57 & 67 & 77 \end{bmatrix}$$

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:

*	*	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 VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo
INTEGER*8       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*8       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*8       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*8       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** ≥ **kd**+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.

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 381 and "F_CHPMV/F_ZHPMV" on page 348 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo
INTEGER*8       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*8       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*8       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*8       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, **n** ≥ 0. If **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 384, and "F_CHPR/F_ZHPR" on page 350 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 VECLIB:

```
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)
```

VECLIB8:

CHARACTER*1 uplo
INTEGER*8 n, incx
REAL*4 alpha, ap(lenap), x(lenx)
CALL SSPR(uplo, n, alpha, x, incx, ap)

CHARACTER*1 uplo
INTEGER*8 n, incx
REAL*8 alpha, ap(lenap), x(lenx)
CALL DSPR(uplo, n, alpha, x, incx, ap)

CHARACTER*1 uplo
INTEGER*8 n, incx
REAL*4 alpha
COMPLEX*8 ap(lenap), x(lenx)
CALL CHPR(uplo, n, alpha, x, incx, ap)

CHARACTER*1 uplo
INTEGER*8 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 $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ 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.

ap Array of length $\mathbf{lenap} = \mathbf{n}\times(\mathbf{n}+1)/2$ 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 386 and "F_CHPR2/F_ZHPR2" on page 352 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 VECLIB:

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)

VECLIB8:

CHARACTER*1 uplo
INTEGER*8 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*8 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*8 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*8 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 $\mathbf{alpha} = 0$, the subprograms do not reference **ap**, **x**, or **y**.	
x	Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times\lvert\mathbf{incx}\rvert+1$ 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 VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     side, uplo
INTEGER*8       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*8       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*8       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*8       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*8       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*8       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** ≥ 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** ≥ 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 389 and “F_CHEMV/F_ZHEMV” on page 342 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo
INTEGER*8       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*8       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*8       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*8       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**, **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**$((i-1)\times$**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 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 392, and “F_CHER/F_ZHER” on page 344 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo
INTEGER*8       n, lda, incx
REAL*4          alpha, a(lda, n), x(lenx)
CALL SSYR(uplo, n, alpha, x, incx, a, lda)
```

```
CHARACTER*1     uplo
INTEGER*8       n, lda, incx
REAL*8          alpha, a(lda, n), x(lenx)
CALL DSYR(uplo, n, alpha, x, incx, a, lda)
```

```
CHARACTER*1     uplo
INTEGER*8       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*8       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 $\mathbf{alpha} = 0$, 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**, $\mathbf{incx} \neq 0$:

		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.
	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

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

$$A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + A,$$

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 394, and "F_CHER2/F_ZHER2" on page 346 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo
INTEGER*8       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*8       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*8       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*8       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, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **a**, **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**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$	x is stored forward in array **x**; that is, x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.

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.

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

VECLIB:

```
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)

VECLIB8:

CHARACTER*1 uplo, trans
INTEGER*8 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*8 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*8 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*8 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*8       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*8       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 $C \leftarrow \alpha A B^T + \bar{\alpha} B A^T + \beta C$
'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, **n** ≥ 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 VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans
INTEGER*8       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*8       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*8       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*8       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*8       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*8 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, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **c**.

k Number of rows or columns in matrix A, **k** ≥ 0, depending on **trans**; refer to description of **A** for details. If **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 397 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 VECLIB:

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)

VECLIB8:

CHARACTER*1 uplo, trans, diag
INTEGER*8 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*8 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*8 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*8 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, **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. 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 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
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 400 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 VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       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*8       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*8       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*8       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

$$x \leftarrow Ax, x \leftarrow A^T x, \quad \text{and} \quad x \leftarrow A^* x.$$

Refer to "F_STPMV/F_DTPMV/F_CTPMV/F_ZTPMV" on page 403 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
REAL*4          ap(lenap), x(lenx)
CALL STPMV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
REAL*8          ap(lenap), x(lenx)
CALL DTPMV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
COMPLEX*8       ap(lenap), x(lenx)
CALL CTPMV(uplo, trans, diag, n, ap, x, incx)
```

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       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 406 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
REAL*4          ap(lenap), x(lenx)
CALL STPSV(uplo, trans, diag, n, ap, x, incx)
```

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
REAL*8          ap(lenap), x(lenx)
CALL DTPSV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*8       n, incx
COMPLEX*8       ap(lenap), x(lenx)
CALL CTPSV(uplo, trans, diag, n, ap, x, incx)

CHARACTER*1     uplo, trans, diag
INTEGER*8       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 $\mathbf{lenap} = \mathbf{n}\times(\mathbf{n}+1)/2$ 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 $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ containing the right-hand-side 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.

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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     side, uplo, transa, diag
INTEGER*8       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*8       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*8       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*8       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, **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 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 408 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       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*8       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*8       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*8       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

$$B \leftarrow \alpha A^{-1}B, \qquad B \leftarrow \alpha A^{-T}B, \qquad B \leftarrow \alpha A^{-*}B,$$

$$B \leftarrow \alpha BA^{-1}, \qquad B \leftarrow \alpha BA^{-T}, \qquad B \leftarrow \alpha BA^{-*}.$$

Refer to “F_STRSM/F_DTRSM/F_CTRSM/F_ZTRSM” on page 414 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     side, uplo, transa, diag
INTEGER*8       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*8       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*8       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*8       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, **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 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 417 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

VECLIB:

```
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)
```

VECLIB8:

```
CHARACTER*1     uplo, trans, diag
INTEGER*8       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*8       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*8       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*8       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

VECLIB:

```
CHARACTER*6     name
INTEGER*4       iarg
CALL XERBLA(name, iarg)
```

VECLIB8:

```
CHARACTER*6     name
INTEGER*8       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 244 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 244 for an example of the storage of symmetric band matrices.

Usage VECLIB

INTEGER*4 **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*4 **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)

VECLIB8

INTEGER*8 **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*8       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.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

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 270 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

VECLIB

```
INTEGER*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

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 275 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 VECLIB

```
INTEGER*4       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*4       INCX, LDA, N, UPLO
REAL*8          ALPHA, BETA
COMPLEX*16      A( LDA, * ), X( * )
SUBROUTINE F_ZHER (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)
```

VECLIB8

```
INTEGER*8       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*8       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 279 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 VECLIB

INTEGER*4 **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*4 **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)

VECLIB8

INTEGER*8 **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*8 **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 249 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 249 for an example of the packed storage of symmetric or Hermitian matrices.

Usage

VECLIB

```
INTEGER*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8 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.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^*$$

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 254 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 VECLIB

```
INTEGER*4       INCX, N, UPLO
REAL*4          ALPHA, BETA
COMPLEX*8       AP( * ), X( * )
SUBROUTINE F_CHPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*4       INCX, N, UPLO
REAL*8          ALPHA, BETA
COMPLEX*16      AP( * ), X( * )
SUBROUTINE F_ZHPR (UPLO, N, ALPHA, X, INCX, BETA, AP)
```

VECLIB8

```
INTEGER*8       INCX, N, UPLO
REAL*4          ALPHA, BETA
COMPLEX*8       AP( * ), X( * )
SUBROUTINE F_CHPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*8       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.

$$A \leftarrow \alpha x x^* + \beta A^* \quad \text{with} \quad A=A^*$$

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 259 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 VECLIB

```
INTEGER*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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.

$$A \leftarrow \alpha x y^* + \bar{\alpha} y x^* + \beta A \quad \text{with} \quad A=A^*$$

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 VECLIB

```
INTEGER*4       CMACH
REAL*4          FUNCTION F_SFPINFO (CMACH)

INTEGER*4       CMACH
REAL*4          FUNCTION F_DFPINFO (CMACH)
```

VECLIB

```
INTEGER*8       CMACH
REAL*4          FUNCTION F_SFPINFO (CMACH)

INTEGER*8       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 173 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:

$$y \leftarrow \alpha A x + \beta y$$

$$y \leftarrow \alpha A^{T} x + \beta y$$

$$y \leftarrow \alpha A^{*} x + \beta y$$

Refer to "SGBMV/DGBMV/CGBMV/ZGBMV" on page 212 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

VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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 $m \le 0$, the subprograms do not reference A, X, or Y.

N Number of columns in matrix A, $n > 0$. If $n \le 0$, 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, $0 \le KL < n$.

	KU	The upper bandwidth of A, that is, the number of nonzero diagonals above the principal diagonal in the band, $0 \le KU < n$.
	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 219 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage VECLIB

```
INTEGER-4       LDA, LDB, M, N, TRANS
REAL*4          A( LDA, * ), B( LDB, * )
SUBROUTINE F_SGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*4       LDA, LDB, M, N, TRANS
REAL*8          A( LDA, * ), B( LDB, * )
SUBROUTINE F_DGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*4       LDA, LDB, M, N, TRANS
COMPLEX*8       A( LDA, * ), B( LDB, * )
SUBROUTINE F_CGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*4       LDA, LDB, M, N, TRANS
COMPLEX*16      A( LDA, * ), B( LDB, * )
SUBROUTINE F_ZGE_COPY(TRANS, M, N, A, LDA, B, LDB)
```

VECLIB8

```
INTEGER*8       LDA, LDB, M, N, TRANS
REAL*4          A( LDA, * ), B( LDB, * )
SUBROUTINE F_SGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*8       LDA, LDB, M, N, TRANS
REAL*8          A( LDA, * ), B( LDB, * )
SUBROUTINE F_DGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*8       LDA, LDB, M, N, TRANS
COMPLEX*8       A( LDA, * ), B( LDB, * )
SUBROUTINE F_CGE_COPY(TRANS, M, N, A, LDA, B, LDB)

INTEGER*8       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 VECLIB

```
INTEGER*4       CONJ, LDA,N
REAL*4          A( LDA, * )
SUBROUTINE F_SGE_TRANS(CONJ, N, A, LDA)

INTEGER*4       CONJ, LDA,N
REAL*8          A( LDA, * )
SUBROUTINE F_DGE_TRANS(CONJ, N, A, LDA)

INTEGER*4       CONJ, LDA,N
COMPLEX*8       A( LDA, * )
SUBROUTINE F_CGE_TRANS(CONJ, N, A, LDA)

INTEGER*4       CONJ, LDA,N
COMPLEX*16      A( LDA, * )
SUBROUTINE F_ZGE_TRANS(CONJ, N, A, LDA)
```

VECLIB8

```
INTEGER*8       CONJ, LDA,N
REAL*4          A( LDA, * )
SUBROUTINE F_SGE_TRANS(CONJ, N, A, LDA)

INTEGER*8       CONJ, LDA,N
REAL*8          A( LDA, * )
SUBROUTINE F_DGE_TRANS(CONJ, N, A, LDA)

INTEGER*8       CONJ, LDA,N
COMPLEX*8       A( LDA, * )
SUBROUTINE F_CGE_TRANS(CONJ, N, A, LDA)

INTEGER*8       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 222 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage VECLIB

INTEGER*4 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*4 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*4 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*4 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)

VECLIB8

INTEGER*8 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*8 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*8 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*8 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 $[0 \ldots m - 1] \times [0 \ldots n - 1]$.

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 232 for a description of the equivalent HP MLIB legacy BLAS subprograms.

Usage VECLIB

INTEGER*4 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*4 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*4 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*4 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)

VECLIB8

INTEGER*8 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*8 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*8 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*8 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, $i = 1,\ldots, n$, 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 VECLIB

INTEGER*4 **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*4 **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*4 **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*4 **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)

VECLIB8

INTEGER*8 **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*8 **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*8 **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*8 **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 VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8 **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*8 **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*8 **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 372 for details. $$x \leftarrow \beta A^T y + z$$
	W	Result of second matrix-vector multiply. Refer to "Purpose" on page 372 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:

$$\text{When } A \in IR^{n^2},\ A \leftarrow \alpha x y^T + \beta A$$

$$\text{When } A \in C^{n^2},\ A \leftarrow \alpha x y^T + \beta A \quad \text{or}$$

$$A \leftarrow \alpha x y^* + \beta A$$

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 237 for a description of HP MLIB legacy BLAS subprograms for rank-1 update.

Usage VECLIB

INTEGER*4 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*4 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*4 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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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 244 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 244 for an example of the storage of symmetric band matrices.

Usage

VECLIB

INTEGER*4 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*4 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*4 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*4 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)

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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. $y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^T$

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 249 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 249 for an example of packed storage for symmetric or Hermitian matrices.

Usage

VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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.

$$y \leftarrow \alpha Ax + \beta y \quad \text{with} \quad A=A^T$$

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 254 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 VECLIB

```
INTEGER*4       INCX, N, UPLO
REAL*4          ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_SSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*4       INCX, N, UPLO
REAL*8          ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_DSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*4       INCX, N, UPLO
COMPLEX*8       ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_CSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*4       INCX, N, UPLO
COMPLEX*16      ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_ZSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)
```

VECLIB8

```
INTEGER*8       INCX, N, UPLO
REAL*4          ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_SSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)
```

```
INTEGER*8       INCX, N, UPLO
REAL*8          ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_DSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*8       INCX, N, UPLO
COMPLEX*8       ALPHA, BETA, AP( * ), X( * )
SUBROUTINE F_CSPR (UPLO, N, ALPHA, X, INCX, BETA, AP)

INTEGER*8       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.

$$A \leftarrow \alpha x x^T + \beta A \quad \text{with} \quad A=A^T$$

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 259 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 VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB*8

```
INTEGER*8       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*8       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*8       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*8       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 270 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

VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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 275 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 VECLIB

```
INTEGER*4       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*4       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*4       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*4       INCX, LDA, N, UPLO
COMPLEX*16      ALPHA, BETA
COMPLEX*16      A( LDA, * ), X( * )
SUBROUTINE F_ZSYR (UPLO, N, ALPHA, X, INCX, BETA, A, LDA)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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 279 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 VECLIB

INTEGER*4 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*4 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*4 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*4 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)

VECLIB8

INTEGER*8 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*8 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*8 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*8 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 294 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 294 for information about the storage of triangular band matrices.

Usage

VECLIB

INTEGER*4 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*4 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*4 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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8       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*8       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 301 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

VECLIB

INTEGER*4 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*4 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*4 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*4 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)

VECLIB8

INTEGER*8 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*8 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*8 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*8 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 308 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 308 for examples that illustrate the packed storage of a triangular matrix.

Usage

VECLIB

```
INTEGER*4       DIAG, INCX, N, TRANS, UPLO
REAL*4          ALPHA
REAL*4          AP( * ), X( * )
SUBROUTINE F_STPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
REAL*8          ALPHA
REAL*8          AP( * ), X( * )
SUBROUTINE F_DTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
COMPLEX*8       ALPHA
COMPLEX*8       AP( * ), X( * )
SUBROUTINE F_CTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
COMPLEX*16      ALPHA
COMPLEX*16      AP( * ), X( * )
SUBROUTINE F_ZTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)
```

VECLIB

```
INTEGER*8       DIAG, INCX, N, TRANS, UPLO
REAL*4          ALPHA
REAL*4          AP( * ), X( * )
SUBROUTINE F_STPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*8       DIAG, INCX, N, TRANS, UPLO
REAL*8          ALPHA
REAL*8          AP( * ), X( * )
SUBROUTINE F_DTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*8       DIAG, INCX, N, TRANS, UPLO
COMPLEX*8       ALPHA
COMPLEX*8       AP( * ), X( * )
SUBROUTINE F_CTPMV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*8       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 313 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 VECLIB

```
INTEGER*4       DIAG, INCX, N, TRANS, UPLO
REAL*4          ALPHA, AP( * ), X( * )
SUBROUTINE F_STPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
REAL*8          ALPHA, AP( * ), X( * )
SUBROUTINE F_DTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
COMPLEX*8       ALPHA, AP( * ), X( * )
SUBROUTINE F_CTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP,  X, INCX)

INTEGER*4       DIAG, INCX, N, TRANS, UPLO
COMPLEX*16      ALPHA, AP( * ), X( * )
SUBROUTINE F_ZTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)
```

VECLIB8

```
INTEGER*8       DIAG, INCX, N, TRANS, UPLO
REAL*4          ALPHA, AP( * ), X( * )
SUBROUTINE F_STPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)
```

```
INTEGER*8        DIAG, INCX, N, TRANS, UPLO
REAL*8           ALPHA, AP( * ), X( * )
SUBROUTINE F_DTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP, X, INCX)

INTEGER*8        DIAG, INCX, N, TRANS, UPLO
COMPLEX*8        ALPHA, AP( * ), X( * )
SUBROUTINE F_CTPSV (UPLO, TRANS, DIAG, N, ALPHA, AP,  X, INCX)

INTEGER*8        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 323 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 VECLIB

```
INTEGER*4       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*4       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*4       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*4 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)

VECLIB8

INTEGER*8 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*8 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*8 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*8 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 VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

```
INTEGER*8       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*8       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*8 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*8 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_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.

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**	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 327 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 VECLIB

```
INTEGER*4       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*4       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*4       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*4       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)
```

VECLIB8

INTEGER*8 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*8 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*8 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*8 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 332 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 VECLIB

INTEGER*4 DIAG, INCX, N, TRANS, UPLO
REAL*4 ALPHA, A(LDA, *), X(*)
SUBROUTINE F_STRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X, INCX)

INTEGER*4 DIAG, INCX, N, TRANS, UPLO
REAL*8 ALPHA, A(LDA, *), X(*)
SUBROUTINE F_DTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X, INCX)

INTEGER*4 DIAG, INCX, N, TRANS, UPLO
COMPLEX*8 ALPHA, A(LDA, *), X(*)
SUBROUTINE F_CTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X, INCX)

INTEGER*4 DIAG, INCX, N, TRANS, UPLO
COMPLEX*16 ALPHA, A(LDA, *), X(*)
SUBROUTINE F_ZTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X, INCX)

VECLIB8

```
INTEGER*8       DIAG, INCX, N, TRANS, UPLO
REAL*4          ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_STRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER*8       DIAG, INCX, N, TRANS, UPLO
REAL*8          ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_DTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA, X,
INCX)

INTEGER*8       DIAG, INCX, N, TRANS, UPLO
COMPLEX*8       ALPHA, A( LDA, * ), X( * )
SUBROUTINE F_CTRSV (UPLO, TRANS, DIAG, N, ALPHA, A, LDA,  X,
INCX)

INTEGER*8       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 BLAS Operations

Overview

This chapter explains how to use the HP MLIB Sparse BLAS implementation that supports matrix multiply and triangular solver.

This implementation provides the NIST Fortran Sparse BLAS V.0.5 standard functionality, but with three additional data types (“Sparse BLAS Naming Convention—Data Type” on page 423) and four additional matrix forms (See MSC, MSR, BMC, and BMR in Table 4-2). The Sparse BLAS library routines are written in Fortran 77 and are callable from Fortran 90 and C routines.

Chapter objectives

After reading this chapter you will:

- Be familiar with the Sparse BLAS subroutine naming convention
- Understand argument conventions
- Know what operations the Sparse BLAS performs
- Know how to use the described subprograms
- Be familiar with the Sparse BLAS subroutines

Associated documentation

The following documents provide supplemental material for this chapter:

Remington, Karin A. *The NIST Fortran Sparse BLAS Library Reference Implementation*. National Institute of Standards and Technology. August, 1997.

Pozo, Roldan, Karin A. Remington. *NIST Sparse BLAS User's Guide*. National Institute of Standards and Technology. July, 1996.

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 Sparse BLAS

Subroutine naming convention

Each Sparse BLAS subroutine has a six-character name that is a function of the data type, the data structure of the sparse matrix and the type of operation. More precisely, each name is in the form XYYYZZ.

The first letter, denoted by X, in the naming convention indicates one of the four Fortran data types, as shown in Table 4-1.

Table 4-1 Sparse 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 three letters in the naming convention, YYY, indicate the data structure of the sparse matrix, as presented in Table 4-2.

Table 4-2 Sparse BLAS Naming Convention—Matrix Form

YYY	Point Entry
COO	Coordinate
CSC	Compressed sparse column
MSC	Modified sparse column
CSR	Compressed sparse row
MSR	Modified sparse row
DIA	Sparse diagonal
ELL	Ellpack-Itpack
JAD	Jagged diagonal
SKY	(Triangular) Skyline
YYY	**Block Entry**
BCO	Block coordinate
BSC	Block compressed sparse column
BMC	Block modified sparse column
BSR	Block compressed sparse row
BMR	Block modified sparse row
BDI	Block sparse diagonal
BEL	Block Ellpack-Itpack
VBR	Variable block row

The table below lists the final two characters in the naming convention, ZZ, indicating the type of operation.

Sparse BLAS Naming Convention—Operation

ZZ	Subroutine Operation
MM	Matrix-Matrix product
SM	Solution of a triangular system with mulitple right-hand-sides

Sparse matrix storage formats

Each Sparse BLAS subroutine handles a specific sparse matrix storage format. These formats are classified into two major groups: point entry and block entry data structures, as presented in Table 4-2. Point entry formats store the nonzero entries and keep track of their location in the matrix. Block entry formats represent sparse matrices whose nonzero entries are dense blocks.

Point entry formats

COO - Coordinate. Given a sparse matrix *A* with *nnz* nonzero entries, the COO format stores the entries of the matrix along with their corresponding row and column indices. Three arrays are required for the COO representation:

- *val(*)* - Scalar array of length *nnz* containing the matrix entries.
- *indx(*)* - Integer array of length *nnz* containing row indices such that *indx(i)* corresponds to the row index of *val(i)*.
- *jndx(*)* - Integer array of length *nnz* containing column indices such that *jndx(i)* corresponds to the column index of *val(i)*.

Consider, for example, the 4 x 5 matrix:

Table 4-3 4 x 5 Matrix

11	0	13	0	15
21	22	0	0	0
31	0	33	0	35
41	0	0	44	45

This matrix could be represented in COO format as:

Table 4-4 COO Format Matrix

val=	11	13	15	21	22	31	33	35	41	44	45
indx=	1	1	1	2	2	3	3	3	4	4	4
jndx=	1	3	5	1	2	1	3	5	1	4	5

CSC - Compressed sparse column. Given a sparse *m-by-k* matrix *A* with *nnz* nonzero entries, the CSC format stores each column of *A* as a sparse vector. Four arrays are required for the CSC representation:

- *val(*)* - Scalar array of length *max(nnz, pntre(k)-1)* containing the nonzero matrix entries.
- *indx(*)* - Integer array of length *max(nnz, pntre(k)-1)* containing row indices such that *indx(i)* corresponds to the row index of *val(i)*.
- *pntrb(*)* - Integer array of length *k* such that *pntrb(j)* points to location in *val( )* of the first nonzero element in column *j*.
- *pntre(*)* - Integer array of length *k* such that *pntre(j) -1* points to location in *val( )* of the last nonzero element in column *j*.

Note that the above representation only requires each column of *A* to be stored in a contiguous portion of *val( )*. If matrix *A* is stored in the first *nnz* entries of

val( ), a single array *pntr( )* of length *k+1* can be used instead of *pntrb( )* and *pntre( )* by having *pntrb(i) = pntr(i)* and *pntr(i) = pntr(i+1)*.

The matrix in Table 4-4 could be represented in CSC format as:

Table 4-5 CSC Format Matrix

val=	11	21	31	41	22	13	33	44	15	35	45
indx=	1	2	3	4	2	1	3	4	1	3	4
pntrb=	1	5	6	8	9						
pntre=	5	6	8	9	12						

Optionally, a single array *pntrb( )* (instead of *pntrb( )* and *pntre( )*) could be used:

pntr=	1	5	6	8	9	12

MSC - Modified sparse column. The MSC format is a variation of the CSC format obtained by storing the main diagonal of the matrix in a specific array. Given a sparse *m-by-k* matrix *A* with *nnz* nonzero entries, the MSC representation requires five arrays:

- *diag(*)* - Scalar array of length *d* containing the main diagonal of *A*, where *d=min(m, k)* is the number of elements in the main diagonal.
- *val(*)* - Scalar array of length *max(nnz-d, pntre(k)-1)* containing the nonzero matrix entries that do not belong to the main diagonal.
- *indx(*)* - Integer array of length *max(nnz, pntre(k)-1)* containing row indices such that *indx(i)* corresponds to the row index of *val(i)*.
- *pntrb(*)* - Integer array of length *k* such that *pntrb(j)* points to location in *val( )* of the first nonzero element in column *j*.
- *pntre(*)* - Integer array of length *k* such that *pntre(j)-1* points to location in *val( )* of the last nonzero element in column *j*.

The matrix in Table 4-4 could be represented in MSC format as:

Table 4-6 MSC Format Matrix

diag=	11	22	33	44			
val=	21	31	41	13	15	35	45
indx=	2	3	4	1	1	3	4
pntrb=	1	4	4	5	5		
pntre=	4	4	5	5	8		

CSR - Compressed sparse row. Given a sparse *m-by-k* matrix *A* with *nnz* nonzero entries, the CSR format stores each row of *A* as a sparse vector. Four arrays are required for the CSR representation:

- *val(*)* - Scalar array of length *max(nnz, pntre(k)-1)* containing the nonzero matrix entries.
- *indx(*)* - Integer array of length *max(nnz, pntre(k)-1)* containing column indices such that *indx(i)* corresponds to the column index of *val(i)*.
- *pntrb(*)* - Integer array of length *m* such that *pntrb(j)* points to location in *val( )* of the first nonzero element in row *j*.
- *pntre(*)* - Integer array of length *m* such that *pntre(j)-1* points to location in *val( )* of the last nonzero element in row *j*.

Note that the above representation only requires each row of *A* to be stored in a contiguous portion of *val( )*. If matrix *A* is stored in the first *nnz* entries of *val( )*, a single array *pntr( )* of length *m+1* can be used instead of *pntrb( )* and *pntre( )* by having *pntrb(i) = pntr(i)* and *pntre(i) = pntr(i+1)*.

The matrix in Table 4-4 could be represented in CSR format as:

Table 4-7 CSR Format Matrix

val=	11	13	15	21	22	31	33	35	41	44	45
indx=	1	3	5	1	2	1	3	5	1	4	5
pntrb=	1	4	6	9							
pntre=	4	6	8	12							

Optionally, a single array *pntr( )* (instead of *pntrb( )* and *pntre( )*) could be used:

pntr=	1	4	6	9	12

MSR - Modified sparse row. The MSR format is a variation of the CSR format obtained by storing the main diagonal of the matrix in a specific array. Given a sparse *m-by-k* matrix *A* with *nnz* nonzero entries, the MSR representation requires five arrays:

- *diag(*)* - Scalar array of length *d* containing the main diagonal of *A*, where *d=min(m, k)* is the number of elements in the main diagonal.
- *val(*)* - Scalar array of length *max(nnz, pntre(k)-1)* containing the nonzero matrix entries that do not belong to the main diagonal.
- *indx(*)* - Integer array of length *max(nnz, pntre(k)-1)* containing column indices such that *indx(i)* corresponds to the column index of *val(i)*.

- *pntrb(*)* - Integer array of length *m* such that *pntrb(j)* points to location in *val( )* of the first nonzero element in row *j*.
- *pntre(*)* - Integer array of length *m* such that *pntre(j)-1* points to location in *val( )* of the last nonzero element in row *j*.

The matrix in Table 4-4 could be represented in MSR format as:

Table 4-8 MSR Format Matrix

diag=	11	22	33	44			
val=	13	15	21	31	35	41	45
indx=	3	5	1	1	5	1	5
pntrb=	1	3	4	6			
pntre=	3	4	6	8			

DIA- Sparse diagonal. Given a sparse *m-by-k* matrix *A* with *ndiag* nonzero diagonals, the DIA format stores the nonzero diagonals of *A*. Two arrays are required for the DIA representation:

- *idiag(*)* - Integer array of length *ndiag* consisting of the corresponding diagonal offsets *idiag(i)* of the nonzero diagonal of *A* stored in the column *val(:, i)*. Offsets are represented with respect to the main diagonal, that is, the main diagonal has offset 0, lower triangular diagonals have negative offsets, and upper triangular diagonals have positive offsets.
- *val(lda,*)* - Two dimensional *lda-by-ndiag* scalar array where *lda* is greater or equal to *min(m, k)*. Column *val(:, i)* consists of elements on diagonal *idiag(i)* of *A* including any zero element within the diagonal. Since diagonals may have fewer elements than to *min(m, k)*:
  - a) any diagonal with positive or zero offset *d= idiag(i)* (the main diagonal or any upper triangular diagonal) is stored starting from position *val(:, i);*
  - b) if *m* is greater or equal to *k*, any diagonal with negative offset *d= idiag(i)* (any lower triangular diagonal) is stored starting from position *val(-d, i);*
  - c) if *m* is less than *k*, any diagonal with negative offset *d= idiag(i)* is stored starting from position *val(t, i)* where *t=1+max(0, k-m-d)*.

Consider, for example, the 5 x 4 matrix:

Table 4-9 5 x 4 Matrix

11	12	0	14
21	22	0	0
0	0	33	34
0	0	0	44
51	52	0	54

This matrix could be represented in DIA format (using *ndiag* = 6) as:

Table 4-10 DIA Format Matrix

idiag=	-4	-3	-1	0	1	3
val=	-	-	21	11	12	14
	-	-	0	22	0	-
	-	0	0	33	34	0
	51	52	54	44	-	-

ELL- Ellpack-Itpack. Given a sparse *m-by-k* matrix *A* with *maxnz* nonzero elements in any row, the ELL format stores the nonzero entries of *A* row by row. Two arrays are required for the ELL representation:

- *val(lda,*)* - Two dimensional *lda-by-maxnz* scalar array where *lda* is greater or equal to *m*. The first entries in row *val(i, :)* consist of nonzero elements in row *i* of *A*.
- *indx(lda, *)* - Two dimensional *lda-by-maxnz* integer array where row *indx(i, :)* stores column indices for row *i* of *A*: *indx(i, j)* corresponds to the column index of *val(i, j)*. If a row has *t* nonzero elements with *t* less than *maxnz*, then *indx(i, t+1)* is set to a negative value.

The matrix in Table 4-10 could be represented in ELL format using *maxn=3*:

Table 4-11 ELL Format Matrix

val=	11	12	14
	21	22	-
	33	34	-
	44	-	-
	51	52	54
indx=	1	2	4
	1	2	-1
	3	4	-1
	4	-1	-
	1	2	4

JAD- Jagged diagonal. Given a sparse *m-by-k* matrix *A* with *maxnz* nonzero elements in any row, the JAD format stores the nonzero entries of a row permutation of *A* using column ordering. The rows of the original matrix *A* are permuted in the decreasing number of nonzero entries. Four arrays are required for the JAD representation:

- *iperm(*)* - Integer array of length *m* such that *i=iperm(j)* indicates that row *j* of the row-permuted representation of *A* corresponds to row *i* of the original matrix *A*. If there is no permutatuion at all, *iperm* is set to a negative number (usually -1).
- *val(*)* - Scalar array of length *nnz* (the total number of nonzero elements in *A*) containing the jagged diagonals of the row-permutation of *A*: The first jagged diagonal consists of the first nonzero entry of each row, and it is stored first in *val()*; the second jagged diagonalconsists of the second nonzero entry of each row, and it is stored second in *val(*)*; and so on.
- *indx(*)* - Integer array of length *nnz* consisting of the column indices of the corresponding entries in *val( )*.
- *pntr(*)* - Integer array of length *maxnz+1* such that *pntr(j)* and *pntr(j+1)-1* respectively point to location in *val( )* of the first and last nonzero elements in the *j*-th jagged diagonal of the row-permutation of *A*.

One possible row-permutation for the matrix in Table 4-10 would be *iperm=(1, 5, 2, 3, 4)*:

Table 4-12 JAD Format Matrix

11	12	0	14
51	52	0	54
21	22	0	0
0	0	33	34
0	0	0	44

And the JAD representation of the row-permuted matrix could be given by:

Table 4-13 JAD Row-Permuted Matrix

val=	11	51	21	33	44	12	52	22	34	14	54
indx=	1	1	1	3	4	2	2	2	4	4	4
pntr=	1	6	10	12							

SKY- (Triangular) Skyline. The Skyline format can be used to represent square triangular matrices. Two arrays are required for the SKY representation:

- *val(*)* - Scalar array of length *pntr(m+1)-1* (see below for *pntr( )*) containing all the nonzero entries, and maybe some zero entries of *A*. *A* must be a square triangular matrix (*m=k*). *val( )* is row oriented if *A* is a lower triangular matrix, and column oriented if *A* is an upper triangular matrix. All entries from the first nonzero entry through the diagonal entry of a row (column) are stored.
- *pntr(*)* - Integer array of length *m+1* (*A* lower triangular) or *k+1* (*A* upper triangular) such that *pntr(i)* and *pntr(i+1)-1*, respectively, point to the location in *val( )* of the first entry and last entry of the skyline profile in row (column) *i*. In any case, the last entry is the diagonal entry.

Consider, for example, the symmetric 5 x 5 matrix:

Table 4-14 5 x 5 Matrix

11	21	0	0	51
21	22	32	42	0
0	32	33	0	0
0	42	0	44	54
51	0	0	54	55

The lower triangular part of this matrix could be represented in SKY format as:

Table 4-15 SKY Format Matrix

val=	11	21	22	32	33	42	0	44	51	0	0	54	55
pntr=	1	2	4	6	9	14							

Block entry formats

All block entry formats can be used to represent matrices with fixed square block size. The Variable Block Compressed Sparse Row format can also be used for matrices with variable block sizes.

BCO - Block coordinate. Given a sparse block matrix *A* formed by *mb-by-kb* square blocks of size *lb-by-lb* each, the block coordinate format represents the *bnnz* nonzero block entries using the same variables as in the COO format. Each nonzero dense block is stored in column major order. Three arrays are required for the BCO representation:

- *val(lb, lb,*)* - Scalar matrix of dimension *lb-by-lb-by-bnnz* containing the nonzero *lb-by-lb* blocks.
- *bindx(*)* - Integer array of length *bnnz* containing block row indices such that *bindx(i)* corresponds to the block row index of the matrix block *val(:, :, i)*.
- *bjndx(*)* - Integer array of length *bnnz* containing block column indices such that *bjndx(i)* corresponds to the block column index of *val(:, :, i)*.

Consider, for example, the 4 x 6 matrix:

Table 4-16 4 x 6 Matrix

11	12	0	0	15	16
21	22	0	0	25	26
0	0	33	0	35	36
0	0	43	44	0	46

This matrix could be represented in BCO format using *mb=2*, *kb=3*, and *lb=2*.

Table 4-17 BCO Format Matrix

bindx=	1	1	2	2
bjndx=	1	3	2	3
val(1:2, 1:2, 1)=	11	12		
	21	22		
val(1:2, 1:2, 2)=	15	16		
	25	26		
val(1:2, 1:2, 3)=	33	0		
	43	44		
val(1:2, 1:2, 4)=	35	36		
	0	46		

BSC - Block compressed sparse column. Given a sparse block matrix *A* formed by *mb-by-kb* square blocks of size *lb-by-lb* each, the block compressed sparse column format represents the *bnnz* nonzero block entries using the same variables as in the CSC format. Each nonzero dense block is stored in column major order. Four arrays are required for the BSC representation:

- *val(lb, lb, *)* - Scalar matrix of dimension *lb-by-lb-by-maxnnz* containing the nonzero *lb-by-lb* blocks, where *maxnnz = max(bnnz, bpntre(k)-1)*.
- *bindx(*)* - Integer array of length *maxnnz* containing block column indices such that *bindx(i)* corresponds to the block column index of *val(:, :, i)*.
- *bpntrb(*)* - Integer array of length *kb* such that *bpntrb(j)* points to location *val(:, :, j)* of the first nonzero block in block column *j*.
- *bpntre(*)* - Integer array of length *kb* such that *pntre(j)-1* points to location *val(:, :, j)* of the last nonzero block in block column *j*.

The matrix in Table 4-17 could be represented in BSC format as:

Table 4-18 BSC Format Matrix

bindx=	1	2	1	2
bpntrb=	1	2	3	
bpntre=	2	3	5	
val(1:2, 1:2, 1)=	11	12		
	21	22		
val(1:2, 1:2, 2)=	33	0		
	43	44		
val(1:2, 1:2, 3)=	15	16		
	25	26		
val(1:2, 1:2, 4)=	35	36		
	0	46		

BMC - Block modified sparse column. The BMC format is a variation of the BSC format obtained by storing the main diagonal of the matrix in a specific array *bdiag*:

- *bdiag(lb, lb, *)* - Scalar matrix of dimension *lb-by-lb-by-d* containing the main diagonal of *A*, where *d=min(mb, kb)* is the number of blocks forming the main diagonal.

BSR - Block compressed sparse row. Given a sparse block matrix *A* formed by *mb-by-kb* square blocks of size *lb-by-lb* each, the block compressed sparse row format represents the *bnnz* nonzero block entries using the same variables as in the BSR format. Each nonzero dense block is stored in column major order. Four arrays are required for the BSR representation:

- *val(lb, lb, *)* - Scalar matrix of dimension *lb-by-lb-by-maxnnz* containing the nonzero *lb-by-lb* blocks, where *maxnnz = max(bnnz, bpntre(k)-1).*
- *bindx(*)* - Integer array of length *maxnnz* containing block row indices such that *bindx(i)* corresponds to the block row index of *val(:, :, i)*.

- *bpntrb(*)* - Integer array of length *mb* such that *bpntrb(j)* points to location *val(:, :, j)* of the first nonzero block in block row *j*.
- *bpntre(*)* - Integer array of length *mb* such that *pntre(j)-1* points to location *val(:, :, j)* of the last nonzero block in block row *j*.

The matrix in Table 4-17 could be represented in BSR format as:

Table 4-19 BSR Format Matrix

bindx=	1	3	2	3
bpntrb=	1	3		
bpntre=	3	4		
val(1:2, 1:2, 1)=	11	12		
	21	22		
val(1:2, 1:2, 2)=	15	16		
	25	26		
val(1:2, 1:2, 3)=	33	0		
	43	44		
val(1:2, 1:2, 4)=	35	36		
	0	46		

BMR - Block modified sparse row. The BMR format is a variation of the BSR format obtained by storing the main diagonal of the matrix in a specific array *bdiag*:

- *bdiag(lb, lb, *)* - Scalar matrix of dimension *lb-by-lb-by-d* containing the main diagonal of *A*, where *d=min(mb, kb)* is the number of blocks forming the main diagonal.

BDI- Block sparse diagonal. Given a sparse block matrix *A* formed by *mb-by-kb* square blocks of size *lb-by-lb* each and with *nbdiag* nonzero block diagonals, the block compressed sparse diagonal format represents the nonzero block diagonals of *A* using the same variables as in the DIA format. Each nonzero dense block is stored in column major order. Two arrays are required for the BDI representation:

- *ibdiag(*)* - Integer array of length *nbdiag* consisting of the corresponding block diagonal offsets *ibdiag(i)* of the nonzero block diagonal of *A* stored in the column *val(:, :, :, i)*.
- *val(lb, lb, lda, *)* - Scalar matrix of dimension *lb-by-lb-by-lda-by-nbdiag* where *lda* is greater or equal to *min(mb, kb)*. Column *val(:, : , :, i)* consists of blocks on block diagonal *ibdiag(i)*.

BEL- Block Ellpack-Itpack. Given a sparse block matrix *A* formed by *mb-by-kb* square blocks of size *lb-by-lb* each and with *maxbnz* nonzero block entries in any block row, the block Ellpack-Itpack format represents the nonzero block entries of *A* using the same variables as in the ELL format. Each nonzero dense block is stored in column major order. Two arrays are required for the BEL representation:

- *val(lb, lb, lda,*)* - Scalar matrix of dimension *lb-by-lb-by-lda-by-maxbnz* where *lda* is greater or equal to *mb*. The first block entries in the block row *val(:, :, i, :)* consist of nonzero blocks in block row *i* of *A*.
- *bindx(*)* - Two dimensional *lda-by-maxbnz* integer array where row *indx(i, :)* stores block column indices for block row *i* of *A*.

VBR- Variable block row. The variable block row format is a generalization of the BSR format. Given a sparse block matrix *A* formed by *mb-by-kb* blocks of size *lb-by-lb* each, the *bnnz* nonzero block of varible sizes are represented by adding arrays *rpntr( )* and *cpntr( )* which store the sparsity structure of the blocks. Each nonzero dense block is stored in column major order. Seven arrays are required for the VBR representation:

- *val(*)* - Scalar array storing nonzero blocks of *A* in row major order.
- *indx(*)* - Integer array of length *bnnz+1* such that *indx(i)* points to the location in *val( )* of the *(1,1)* element of the *i*-th block entry. *indx(bnnz+1)* points to the last used position in *val( )* plus one.
- *bindx(*)* - Integer array of length *bnnz* containing block row indices such that *bindx(i)* corresponds to the block row index of the *i*-th block entry.
- *rpntr(*)* -Integer array of length *mb+1* such that *rpntr(i)* and *rpntr(i+1)*, respectively, are the row index of the first point row and row index of the last point row in the *i*-th block row. Thus, the number of point rows in the *i*-th block row is *rpntr(i+1)-rpntr(i)*.
- *cpntr(*)* - Integer array of length *kb+1* such that *cpntr(j)* and *cpntr(j+1)*, respectively, are the column index of the first point column and the column index of the last point column in the *j*-th block column. Thus, the number of point columns in the *j*-th block column is *cpntr(j+1)-cpntr(j)*.

- *bpntrb(*)* - Integer array of length *mb* such that *bpntrb(j)* points to location in *bindx(*)* of the first nonzero block in block row *j*.
- *bpntre(*)* - Integer array of length *mb* such that *pntre(j)-1* points to location *bindx(*)* of the last nonzero block in block row *j*.

Consider, for example, the 6 x 6 matrix:

Table 4-20 6 x 6 Matrix

11	12	0	14	15	16
21	22	0	24	25	26
31	32	33	0	0	0
0	0	43	44	0	0
0	0	53	54	55	0
0	0	63	64	65	66

This matrix could be represented in VBR format using a 2x1x3 blocking:

Table 4-21 VBR Format Matrix

val=	11	21	12	22	14	24	15	25	16
	26	31	32	33	43	53	63	44	54
	64	8	55	65	0	0	66		
indx=	1	5	11	13	14	17	26		
bindx=	1	3	1	2	2	3			
rpntr=	1	3	4	7					
cpntr=	1	3	4	7					
bpntbr=	1	3	5						
bpntbe=	3	5	7						

Operator arguments in the Sparse BLAS

The argument conventions follow the spirit of the dense BLAS.

- Point entry matrix-matrix product:

  XYYYMM (TRANSA, M, N, K, ALPHA, args(A), B, LDB, BETA, C, LDC, WORK, LWORK)

- Block entry matrix-matrix product:

  XYYYMM (TRANSA, MB, N, KB, ALPHA, args(A), B, BLDB, BETA, C, BLDC, WORK, LWORK)

- Point entry solution of triangular system:

XYYYSM (TRANSA, M, N, UNITD, DV, ALPHA, args(A), B, LDB, BETA, C, LDC, WORK, LWORK)

- Block entry solution of triangular system:

  XYYYSM (TRANSA, MB, N, UNITD, DV, ALPHA, args(A), B, BLDB, BETA, C, BLDC, WORK, LWORK)

The general order of arguments is:

1. Arguments specifying options.
2. Arguments specifying problem dimensions.
3. Input scalar associated with input matrices.
4. Description of sparse input matrices (See “Order of arguments for args(A)” on page 440).
5. Description of dense input matrices.
6. Input scalar associated with input-output matrix.
7. Description of input-output matrix.
8. Workspace.
9. Length of workspace.

Common arguments

The argument TRANSA is an integer argument. The possible values are:

- TRANSA - Indicates how to operate with the sparse matrix.
  - 0 - Operate with the matrix (No transpose).
  - 1 - Operate with the transpose of the matrix.
  - 2 - Operate with the conjugate transpose of the matrix.

  TRANSA=2 is equivalent to TRANSA=1 if the matrix is real.

The argument M is the number of rows in the matrix C, N is the number of columns in C, and K is the number of rows in B. M and K are the row and column dimensions of A, respectively, if TRANSA=0 or the column and row dimensions of A^T or A* if TRANSA=1 or 2.

If A is a constant block entry matrix, then M=MBxLB and K=KBxLB are the row and column dimensions of A, respectively. Here, LB is the block entry dimension given by args(A).

For the block entry data structures, the argument MB is the number of block rows in the matrix C, and KB is the number of block rows in B. MB and KB are

the block row and block column dimensions of A, respectively, if TRANSA=0 or the block column and block row dimensions of A^T or A* if TRANSA=1 or 2.

Negative dimensions are not allowed. If M, N, K, MB, or KB is equal to zero, then no operations are performed.

The arguments ALPHA and BETA are both scalar inputs of the same data type as the matrices.

The arguments B and C are rectangular arrays with first dimension LDB and LDC, respectively.

The argument WORK is an array of the same data type as A and length LWORK. If necessary, it is used as scratch space for optimizing performance.

SM arguments

The argument UNITD is an integer argument indicating whether or not the diagonal matrix D is unitary. The possible values for UNITD are:

1. Unitary diagonal matrix, i.e., D is the identity. In this case the argument DV is ignored.
2. Scale on the left (row scaling).
3. Scale on the right (column scaling).

The argument DV is an array containing the diagonal entries of the (block) diagonal matrix D.

Order of arguments for args(A)

The list of arguments describing the sparse matrix A (denoted by args(A)) is specific to each storage format. The general order of args(A) is:

1. Descriptor array, DESCRA.
2. Array containing the nonzero values (entries) of input matrix.
3. First dimension of values array (if shape of this array differs from index arrays).
4. Array(s) containing indices corresponding to the entries of input matrix. If more than one index array:
   a. Arrays ordered in decreasing length.
   b. Arrays of same length are ordered with row indices first.
5. Length or first dimension of value/index arrays.
6. Array(s) containing pointers. If more than one pointer array:
   a. Arrays ordered in decreasing length.
   b. Arrays of same length are ordered with row pointers first, column pointers next, and diagonal pointers last.
7. Length or first dimension of pointer arrays.
8. Array(s) representing left and/or right permutations of A. If more than one permutation array order left permutation array first.
9. Block entry dimension, LB.

Sparse BLAS routines

Name SBCOMM/DBCOMM/CBCOMM/ZBCOMM
Block coordinate matrix-matrix multiply

Purpose Block coordinate matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with m=mb x lb and k=kb x lb. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SBCOMM
INTEGER*4 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bjndx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBCOMM
INTEGER*4 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bjndx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBCOMM
INTEGER*4 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bjndx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBCOMM
INTEGER*4 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bjndx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBCOMM
INTEGER*8 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bjndx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBCOMM
INTEGER*8 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bjndx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBCOMM
INTEGER*8 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bjndx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBCOMM
INTEGER*8 transa, mb, n, kb, bnnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bjndx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBCOMM (transa, mb, n, kb, alpha, descra, val, bindx, bjndx, bnnz, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

kb Number of block columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Scalar array of length *nnz* containing matrix entries stored column-major within each dense block.

bindx( ) Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.

bjndx( )	Integer array of length *bnnz* consisting of the block column indices of the block entries of *A*.
bnnz	Number of block entries.
lb	Dimension of dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SBDIMM/DBDIMM/CBDIMM/ZBDIMM
Block diagonal matrix-matrix multiply

Purpose Block diagonal matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with m=mb x lb and k=kb x lb. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SBDIMM
INTEGER*4 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBDIMM
INTEGER*4 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBDIMM
INTEGER*4 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBDIMM
INTEGER*4 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBDIMM
INTEGER*8 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBDIMM
INTEGER*8 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBDIMM
INTEGER*8 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBDIMM
INTEGER*8 transa, mb, n, kb, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBDIMM (transa, mb, n, kb, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

	1: Operate with transpose matrix 2: Operate with conjugate-transpose matrix
mb	Number of block rows in matrix *A*.
n	Number of columns in matrix *C*.
kb	Number of block columns in matrix *A*.
alpha	Scalar parameter.
descra( )	Descriptor argument. Five element integer array.
descra(1)	Matrix structure. 0: General 1: Symmetric 2: Hermitian 3: Triangular 4: Skew (Anti)-Symmetric 5: Diagonal
descra(2)	Upper/Lower triangular indicator. 1: Lower 2: Upper
descra(3)	Main diagonal type. 0: Non-unit 1: Unit
descra(4)	Array base. 0: C/C++ compatible Not Supported 1: Fortran compatible
descra(5)	Repeated indices. 0: Unknown 1: No repeated indices
val( )	Scalar array of length *lb*lb*blda*nbdiag* containing matrix entries stored column-major within each dense block.
blda	Leading block dimension of *val( )*.

ibdiag( )	Integer array of length *nbdiag* consisting of the corresponding indices of the nonzero block diagonals of *A* in *val( )*.
nbdiag	Number of nonzero block diagonals in *A*.
lb	Row and column dimension of the dense blocks composing *val*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name

SBDISM/DBDISM/CBDISM/ZBDISM
Block diagonal format triangular solve

Purpose

Block diagonal format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B with m=mb x lb, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage

VECLIB:

SUBROUTINE SBDISM
INTEGER*4 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBDISM
INTEGER*4 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBDISM
INTEGER*4 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBDISM
INTEGER*4 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*4 descra(*), ibdiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBDISM
INTEGER*8 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBDISM
INTEGER*8 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBDISM
INTEGER*8 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBDISM
INTEGER*8 transa, mb, n, unitd, blda, nbdiag, lb, ldb, ldc, lwork
INTEGER*8 descra(*), ibdiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBDISM (transa, mb, n, unitd, dv, alpha, descra, val, blda, ibdiag, nbdiag, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *lb*lb*mb*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *lb*lb*blda*nbdiag* containing matrix entries stored column-major within each dense block.
blda	Leading block dimension of *val*( ).
ibdiag( )	Integer array of length *nbdiag* consisting of the corresponding indices of the nonzero block diagonals of *A* in *val( )*.
nbdiag	Number of nonzero block diagonals in *A*.
lb	Dimension of the dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SBELMM/DBELMM/CBELMM/ZBELMM
Block Ellpack matrix-matrix multiply

Purpose Block Ellpack matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with m=mb x lb and k=kb x lb. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SBELMM
INTEGER*4 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBELMM
INTEGER*4 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBELMM
INTEGER*4 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBELMM
INTEGER*4 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBELMM
INTEGER*8 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBELMM
INTEGER*8 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBELMM
INTEGER*8 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBELMM
INTEGER*8 transa, mb, n, kb, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBELMM (transa, mb, n, kb, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

kb Number of block columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Scalar array of length *lb*lb*blda*maxbnz* containing matrix entries stored column-major within each dense block.

blda Leading block dimension of *bindx(:,:)*.

bindx( )	Two dimensional *blda-by-maxbnz* array such that *bindx (i, :)* consists of the block column indices of the nonzero blocks in block row *i*, padded by the integer value *i* if the number of nonzero blocks is less than *maxbnz*.
maxbnz	Max number of nonzero blocks per row.
lb	Row and column dimension of the dense blocks composing *val*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SBELSM/DBELSM/CBELSM/ZBELSM
Block Ellpack format triangular solve

Purpose Block Ellpack format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B with m=mb x lb, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE SBELSM
INTEGER*4 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, naxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBELSM
INTEGER*4 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBELSM
INTEGER*4 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBELSM
INTEGER*4 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBELSM
INTEGER*8 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBELSM
INTEGER*8 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBELSM
INTEGER*8 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBELSM
INTEGER*8 transa, mb, n, unitd, blda, maxbnz, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBELSM (transa, mb, n, unitd, dv, alpha, descra, val, blda, bindx, maxbnz, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *lb*lb*mb*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *lb*lb*blda*maxbnz* containing matrix entries stored column-major within each dense block.
blda	Leading block dimension of *bindx(:,:)*.
bindx( )	Two dimensional *blda-by-maxbnz* array such that *bindx (i,:)* consists of the block column indices of the nonzero blocks in block row *i*, padded by the integer value *i* if the number of nonzero blocks is less than *maxbnz*.
maxbnz	Max number of nonzero blocks per row.
lb	Row and column dimension of the dense blocks composing *val*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SBSCMM/DBSCMM/CBSCMM/ZBSCMM
Block sparse column matrix-matrix multiply

Purpose Block sparse column matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with m=mb x lb and k=kb x lb. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SBSCMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSCMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSCMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSCMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBSCMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSCMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSCMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSCMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSCMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

kb Number of block columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Scalar array of length *nnz* containing matrix entries stored column-major within each dense block.

bindx( ) Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.

bpntrb( )	Integer array of length *kb* such that *bpntrb(j)* points to location in *bindx* of the first block entry of the *j*-th block column of *A*.
bpntre( )	Integer array of length *kb* such that *bpntre(j)-1* points to location in *bindx* of the last block entry of the *j*-th block column of *A*.
lb	Dimension of dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SBSCSM/DBSCSM/CBSCSM/ZBSCSM
Block sparse column format triangular solve

Purpose Block sparse column format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B with m=mb x lb, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE SBSCSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSCSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSCSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSCSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBSCSM
INTEGER*8 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSCSM
INTEGER*8 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSCSM
INTEGER*8 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSCSM
INTEGER*8 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSCSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *lb*lb*mb*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *nnz* containing matrix entries stored column-major within each dense block.
bindx( )	Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.
bpntrb( )	Integer array of length *kb* such that *bpntrb(j)* points to location in *bindx* of the first block entry of the *j*-th block column of *A*.
bpntre( )	Integer array of length *kb* such that *bpntre(j)-1* points to location in *bindx* of the last block entry of the *j*-th block column of *A*.
lb	Dimension of the dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SBSRMM/DBSRMM/CBSRMM/ZBSRMM
Block sparse row matrix-matrix multiply

Purpose Block sparse row matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with m=mb x lb and k=kb x lb. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SBSRMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSRMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSRMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSRMM
INTEGER*4 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SBSRMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSRMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSRMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZBSRMM
INTEGER*8 transa, mb, n, kb, lb, ldb, ldc, lwork
INTEGER*8 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSRMM (transa, mb, n, kb, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

kb Number of block columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Scalar array of length *nnz* containing matrix entries stored column-major within each dense block.

bindx( ) Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.

bpntrb( )	Integer array of length *mb* such that *bpntrb(j)* points to location in *bindx* of the first block entry of the *j*-th block row of *A*.
bpntre( )	Integer array of length *mb* such that *bpntre(j)-1* points to location in *bindx* of the last block entry of the *j*-th block column of *A*.
lb	Dimension of dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name

SBSRSM/DBSRSM/CBSRSM/ZBSRSM
Block sparse row format triangular solve

Purpose

Block sparse row format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B with m=mb x lb, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage

VECLIB:

SUBROUTINE SBSRSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DBSRSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CBSRSM
INTEGER*4 transa, mb, n, unitd, blda, lb, ldb, ldc, lwork
INTEGER*4 descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx, bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)

```
SUBROUTINE      ZBSRSM
INTEGER*4       transa, mb, n, unitd, blda,  lb, ldb, ldc, lwork
INTEGER*4       descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16      alpha, beta
COMPLEX*16      val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx,
bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)
```

VECLIB8:

```
SUBROUTINE      SBSRSM
INTEGER*8       transa, mb, n, unitd, blda,  lb, ldb, ldc, lwork
INTEGER*8       descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*4          alpha, beta
REAL*4          val(*), b(ldb,*), c(ldc,*), work(*)
CALL SBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx,
bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      DBSRSM
INTEGER*8       transa, mb, n, unitd, blda,  lb, ldb, ldc, lwork
INTEGER*8       descra(*), bindx(*), bpntrb(*), bpntre(*)
REAL*8          alpha, beta
REAL*8          val(*), b(ldb,*), c(ldc,*), work(*)
CALL DBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx,
bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      CBSRSM
INTEGER*8       transa, mb, n, unitd, blda,  lb, ldb, ldc, lwork
INTEGER*8       descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*8       alpha, beta
COMPLEX*8       val(*), b(ldb,*), c(ldc,*), work(*)
CALL CBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx,
bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      ZBSRSM
INTEGER*8       transa, mb, n, unitd, blda,  lb, ldb, ldc, lwork
INTEGER*8       descra(*), bindx(*), bpntrb(*), bpntre(*)
COMPLEX*16      alpha, beta
COMPLEX*16      val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZBSRSM (transa, mb, n, unitd, dv, alpha, descra, val, bindx,
bpntrb, bpntre, lb, b, ldb, beta, c, ldc, work, lwork)
```

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *lb*lb*mb*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *nnz* containing matrix entries stored column-major within each dense block.
bindx( )	Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.
bpntrb( )	Integer array of length *mb* such that *bpntrb(j)* points to location in *bindx* of the first block entry of the *j*-th block row of *A*.
bpntre( )	Integer array of length *mb* such that *bpntre(j)-1* points to location in *bindx* of the last block entry of the *j*-th block row of *A*.
lb	Dimension of the dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name

SCOOMM/DCOOMM/CCOOMM/ZCOOMM
Coordinate matrix-matrix multiply

Purpose

Coordinate matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage

VECLIB:

SUBROUTINE SCOOMM
INTEGER*4 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), jndx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCOOMM
INTEGER*4 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), jndx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCOOMM
INTEGER*4 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), jndx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCOOMM
INTEGER*4 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), jndx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SCOOMM
INTEGER*8 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), jndx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCOOMM
INTEGER*8 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), jndx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCOOMM
INTEGER*8 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), jndx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCOOMM
INTEGER*8 transa, m, n, k, nnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), jndx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCOOMM (transa, m, n, k, alpha, descra, val, indx, jndx, nnz, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

k Number of columns in matrix *A*.

alpha Scalar parameter.

descra( )
Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Scalar array of length *nnz* containing matrix entries.

indx( ) Integer array of length *nnz* containing row indices.

jndx( ) Integer array of length *nnz* containing column indices.

nnz( ) Number of nonzero elements in *A*.

b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SCSCMM/DCSCMM/CCSCMM/ZCSCMM
Compressed sparse column matrix-matrix multiply

Purpose Compressed sparse column matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-*by*-k sparse matrix, and B is a k-*by*-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-*by*-m matrix, and B is a m-*by*-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SCSCMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSCMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSCMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

```
SUBROUTINE      ZCSCMM
INTEGER*4       transa, m, n, k, ldb, ldc, lwork
INTEGER*4       descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16      alpha, beta
COMPLEX*16      val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc, work, lwork)
```

VECLIB8:

```
SUBROUTINE      SCSCMM
INTEGER*8       transa, m, n, k, ldb, ldc, lwork
INTEGER*8       descra(*), indx(*), pntrb(*), pntre(*)
REAL*4          alpha, beta
REAL*4          val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      DCSCMM
INTEGER*8       transa, m, n, k, ldb, ldc, lwork
INTEGER*8       descra(*), indx(*), pntrb(*), pntre(*)
REAL*8          alpha, beta
REAL*8          val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      CCSCMM
INTEGER*8       transa, m, n, k, ldb, ldc, lwork
INTEGER*8       descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8       alpha, beta
COMPLEX*8       val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc, work, lwork)
```

```
SUBROUTINE      ZCSCMM
INTEGER*8       transa, m, n, k, ldb, ldc, lwork
INTEGER*8       descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16      alpha, beta
COMPLEX*16      val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSCMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc, work, lwork)
```

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

	1: Operate with transpose matrix 2: Operate with conjugate-transpose matrix
m	Number of rows in matrix *A*.
n	Number of columns in matrix *C*.
k	Number of columns in matrix *A*.
alpha	Scalar parameter.
descra( )	Descriptor argument. Five element integer array.
descra(1)	Matrix structure. 0: General 1: Symmetric 2: Hermitian 3: Triangular 4: Skew (Anti)-Symmetric 5: Diagonal
descra(2)	Upper/Lower triangular indicator. 1: Lower 2: Upper
descra(3)	Main diagonal type. 0: Non-unit 1: Unit
descra(4)	Array base. 0: C/C++ compatible Not Supported 1: Fortran compatible
descra(5)	Repeated indices. 0: Unknown 1: No repeated indices
val( )	Scalar array of length *nnz* containing matrix entries.
indx( )	Integer array of length *nnz* containing row indices.
pntrb( )	Integer array of length *k* such that *pntrb(j)* points to location in *val* of the first nonzero element in column *j*.

pntre( )	Integer array of length *k* such that *pntre(j)-1* points to location in *val* of the last nonzero element in column *j*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SCSCSM/DCSCSM/CCSCSM/ZCSCSM
Compressed sparse column format triangular solve

Purpose Compressed sparse column format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE SCSCSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSCSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSCSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSCSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SCSCSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSCSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSCSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSCSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSCSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *nnz* containing matrix entries.
indx( )	Integer array of length *nnz* containing row indices.
pntrb( )	Integer array of length *k* such that *pntrb(j)* points to location in *val* of the first nonzero element in column *j*.
pntre( )	Integer array of length *k* such that *pntre(j)-1* points to location in *val* of the first nonzero element in column *j*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SCSRMM/DCSRMM/CCSRMM/ZCSRMM
Compressed sparse row matrix-matrix multiply

Purpose Compressed sparse row matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SCSRMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSRMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSRMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSRMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SCSRMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSRMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSRMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSRMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSRMM (transa, m, n, k, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

	1: Operate with transpose matrix 2: Operate with conjugate-transpose matrix
m	Number of rows in matrix *A*.
n	Number of columns in matrix *C*.
k	Number of columns in matrix *A*.
alpha	Scalar parameter.
descra( )	Descriptor argument. Five element integer array.
descra(1)	Matrix structure. 0: General 1: Symmetric 2: Hermitian 3: Triangular 4: Skew (Anti)-Symmetric 5: Diagonal
descra(2)	Upper/Lower triangular indicator. 1: Lower 2: Upper
descra(3)	Main diagonal type. 0: Non-unit 1: Unit
descra(4)	Array base. 0: C/C++ compatible Not Supported 1: Fortran compatible
descra(5)	Repeated indices. 0: Unknown 1: No repeated indices
val( )	Scalar array of length *nnz* containing matrix entries.
indx( )	Integer array of length *nnz* containing column indices.
pntrb( )	Integer array of length *m* such that *pntrb(j)* points to location in *val* of the first nonzero element in row *j*.

pntre( )	Integer array of length *m* such that *pntre(j)-1* points to location in *val* of the last nonzero element in row *j*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SCSRSM/DCSRSM/CCSRSM/ZCSRSM
Compressed sparse row format triangular solve

Purpose Compressed sparse row format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE **SCSRSM**
INTEGER*4 **transa, m, n, unitd, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), pntrb(*), pntre(*)**
REAL*4 **alpha, beta**
REAL*4 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL SCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **DCSRSM**
INTEGER*4 **transa, m, n, unitd, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), pntrb(*), pntre(*)**
REAL*8 **alpha, beta**
REAL*8 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL DCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **CCSRSM**
INTEGER*4 **transa, m, n, unitd, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), pntrb(*), pntre(*)**
COMPLEX*8 **alpha, beta**
COMPLEX*8 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL CCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSRSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SCSRSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DCSRSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CCSRSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZCSRSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntrb(*), pntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZCSRSM (transa, m, n, unitd, dv, alpha, descra, val, indx, pntrb, pntre, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Scalar array of length *nnz* containing matrix entries.
indx( )	Integer array of length *nnz* containing column indices.
pntrb( )	Integer array of length *m* such that *pntrb(j)* points to location in *val* of the first nonzero element in row *j*.
pntre( )	Integer array of length *m* such that *pntre(j)-1* points to location in *val* of the first nonzero element in row *j*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SDIAMM/DDIAMM/CDIAMM/ZDIAMM
Diagonal matrix-matrix multiply

Purpose Diagonal matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SDIAMM
INTEGER*4 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DDIAMM
INTEGER*4 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CDIAMM
INTEGER*4 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZDIAMM
INTEGER*4 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SDIAMM
INTEGER*8 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DDIAMM
INTEGER*8 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CDIAMM
INTEGER*8 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZDIAMM
INTEGER*8 transa, m, n, k, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZDIAMM (transa, m, n, k, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

k Number of columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Two dimensional *lda-by-ndiag* array such that *val(:, i)* consists of nonzero elements on diagonal *idiag(i)* of *A*. Diagonals in the lower triangular part of *A* are padded from the top, and those in the upper triangular part are padded from the bottom.

lda	Leading dimension of *val*, must be greater or equal to *min (m,k)*.
idiag( )	Integer array of length *ndiag* consisting of the corresponding diagonal offsets of the nonzero diagonals of *A* in *val*. Lower triangular diagonals have negative offsets, the main diagonal has offset 0, and upper triangular diagonals have positive offset.
ndiag	Number of nonzero diagonals in *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SDIASM/DDIASM/CDIASM/ZDIASM
Diagonal format triangular solve

Purpose Diagonal format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by $A^{-}*$. Here, A^{-T} is the transpose-inverse and $A^{-}*$ is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-}*B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-}*B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-}*DB + \beta C$$

Usage VECLIB:

SUBROUTINE SDIASM
INTEGER*4 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DDIASM
INTEGER*4 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CDIASM
INTEGER*4 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZDIASM
INTEGER*4 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*4 descra(*), idiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SDIASM
INTEGER*8 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DDIASM
INTEGER*8 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CDIASM
INTEGER*8 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZDIASM
INTEGER*8 transa, m, n, unitd, lda, ndiag, ldb, ldc, lwork
INTEGER*8 descra(*), idiag(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZDIASM (transa, m, n, unitd, dv, alpha, descra, val, lda, idiag, ndiag, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Two dimensional *lda-by-ndiag* array such that *val(:, i)* consists of nonzero elements on diagonal *idiag(i)* of *A*. Diagonals in the lower triangular part of *A* are padded from the top, and those in the upper triangular part are padded from the bottom.
lda	Leading dimension of *val*, must be greater or equal to *min(m,k)*.
idiag( )	Integer array of length *ndiag* consisting of the corresponding diagonal offsets of the nonzero diagonals of *A* in *val*. Lower triangular diagonals have negative offsets, the main diagonal has offset 0, and upper triangular diagonals have positive offset.
ndiag	Number of nonzero diagonals in *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SELLMM/DELLMM/CELLMM/ZELLMM
Ellpack matrix-matrix multiply

Purpose Ellpack matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SELLMM
INTEGER*4 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DELLMM
INTEGER*4 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CELLMM
INTEGER*4 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZELLMM
INTEGER*4 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SELLMM
INTEGER*8 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DELLMM
INTEGER*8 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CELLMM
INTEGER*8 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZELLMM
INTEGER*8 transa, m, n, k, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZELLMM (transa, m, n, k, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

k Number of columns in matrix *A*.

alpha Scalar parameter.

descra( )
Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Two dimensional *lda-by-maxnz* array such that *val(i, :)* consists of nonzero elements in row i of A, padded by zero values if the row contains less than *maxnz*.

lda Leading dimension of *val* and *indx*.

indx( )	Two dimensional integer *blda-by-maxbnz* array such that *indx (i, :)* consists of the column indices of the nonzero elements in row *i*, padded by the integer value *i* if the number of nonzeros is less than *maxnz*.
maxnz	Max number of nonzero elements per row.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SELLSM/DELLSM/CELLSM/ZELLSM
Ellpack format triangular solve

Purpose Ellpack format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE SELLSM
INTEGER*4 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DELLSM
INTEGER*4 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CELLSM
INTEGER*4 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZELLSM
INTEGER*4 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SELLSM
INTEGER*8 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DELLSM
INTEGER*8 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CELLSM
INTEGER*8 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZELLSM
INTEGER*8 transa, m, n, unitd, lda, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZELLSM (transa, m, n, unitd, dv, alpha, descra, val, lda, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Two dimensional *lda-by-maxnz* array such that *val(i, :)* consists of nonzero elements in row *i* of *A*, padded by zero values if the row contains less than *maxnz*.
lda	Leading dimension of *val* and *indx*.
indx( )	Two dimensional integer *blda-by-maxnz* array such that *indx(i, :)* consists of the column indices of the nonzero elements in row *i*, padded by the integer value *i* if the number of nonzeros is less than *maxnz*.
maxnz	Max number of nonzero elements per row.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SJADMM/DJADMM/CJADMM/ZJADMM
Jagged diagonal matrix-matrix multiply

Purpose Jagged diagonal matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SJADMM
INTEGER*4 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DJADMM
INTEGER*4 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CJADMM
INTEGER*4 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZJADMM
INTEGER*4 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SJADMM
INTEGER*8 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DJADMM
INTEGER*8 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CJADMM
INTEGER*8 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZJADMM
INTEGER*8 transa, m, n, k, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZJADMM (transa, m, n, k, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

k Number of columns in matrix *A*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Array of length *nnz* (the total number of nonzero entries in *A*) containing the jagged diagonals of the row-permuted representation of *A*. (The row-permutations are performed such that the number of nonzero entries in each row is decreasing.) The first

	jagged diagonal consists of the first nonzero entry of each row, and it is stored first in *val(*)*; the second jagged diagonal consists of the second nonzero entry of each row, and it is stored second in *val(*)*; and so on.
indx( )	Array of length *nnz* consisting of the column indices of the corresponding entries in *val*.
pntr( )	Array of length *maxnz+1*, such that *pntr(i)* and *pntr(i+1)-1,* respectively, point to the location in *val* of the first and last entries of the *i*-th jagged diagonal of the row-permuted representation of *A*.
iperm( )	Array of length *m* such that *i=iperm(j)* indicates that row *j* of the row-permuted representation of *A* corresponds to row *i* of the original matrix *A*. If there is no permutation at all, let *iperm(1)* be a negative number (usually -1).
maxnz	Max number of nonzero elements per row.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name

SJADSM/DJADSM/CJADSM/ZJADSM
Jagged diagonal format triangular solve

Purpose

Jagged diagonal format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by $A^{-}*$. Here, A^{-T} is the transpose-inverse and $A^{-}*$ is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-}*B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-}*B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-}*DB + \beta C$$

Usage

VECLIB:

SUBROUTINE SJADSM
INTEGER*4 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DJADSM
INTEGER*4 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CJADSM
INTEGER*4 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZJADSM
INTEGER*4 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SJADSM
INTEGER*8 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DJADSM
INTEGER*8 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CJADSM
INTEGER*8 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZJADSM
INTEGER*8 transa, m, n, unitd, maxnz, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), pntr(*), iperm(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZJADSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, iperm, indx, maxnz, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Array of length *nnz* (the total number of nonzero entries in *A*) containing the jagged diagonals of the row-permuted representation of *A*. (The row-permutations are performed such that the number of nonzero entries in each row is decreasing.) The first jagged diagonal consists of the first nonzero entry of each row, and it is stored first in *val(*)*; the second jagged diagonal consists of the second nonzero entry of each row, and it is stored second in *val(*)*; and so on.
indx( )	Array of length *nnz* consisting of the column indices of the corresponding entries in *val*.
pntr( )	Array of length *maxnz+1*, such that *pntr(i)* and *pntr(i+1)-1,* respectively, point to the location in *val* of the first and last entries of the *i*-th jagged diagonal of the row-permuted representation of *A*.
iperm( )	Array of length *m* such that *i=iperm(j)* indicates that row *j* of the row-permuted representation of *A* corresponds to row *i* of the original matrix *A*. If there is no permutation at all, let *iperm(1)* be a negative number (usually -1).
maxnz	Max number of nonzero elements per row.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SSKYMM/DSKYMM/CSKYMM/ZSKYMM
Skyline matrix-matrix multiply

Purpose Skyline matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE SSKYMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DSKYMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CSKYMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZSKYMM
INTEGER*4 transa, m, n, k, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SSKYMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DSKYMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CSKYMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZSKYMM
INTEGER*8 transa, m, n, k, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZSKYMM (transa, m, n, k, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

k Number of columns in matrix *A*.

alpha Scalar parameter.

descra( )
Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( ) Array of length *pntr(m+1)-1* (see below for *pntr*) containing all the nonzero entries, and maybe some zero entries of *A*. *A* must be a square triangular matrix (*m=k*). *val( )* is row-oriented if *A* is a lower triangular matrix (*descra(2)=1*) and column oriented if *A* is an

	upper triangular matrix (*descra(2)=2*). All entries from the first nonzero entry through the diagonal entry of a row (column) are stored.
pntr( )	Array of length *m+1* (A lower triangular) or *k+1* (A upper triangular) such that *pntr(i)* and *pntr(i)+1-1,* respectively, point to the location in *val* of the first entry and last entry of the Skyline profile in row (column) *i*. In any case, the last entry is the diagonal entry.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name

SSKYSM/DSKYSM/CSKYSM/ZSKYSM
Skyline format triangular solve

Purpose

Skyline format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by $A^{-}*$. Here, A^{-T} is the transpose-inverse and $A^{-}*$ is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-}*B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-}*B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-}*DB + \beta C$$

Usage

VECLIB:

SUBROUTINE SSKYSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)

CALL SSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DSKYSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)

CALL DSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CSKYSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)

CALL CSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZSKYSM
INTEGER*4 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*4 descra(*), pntr(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SSKYSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DSKYSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CSKYSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZSKYSM
INTEGER*8 transa, m, n, unitd, ldb, ldc, lwork
INTEGER*8 descra(*), pntr(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZSKYSM (transa, m, n, unitd, dv, alpha, descra, val, pntr, b. ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix

1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

m Number of rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *M*.

alpha Scalar parameter.

descra( )

Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3) Main diagonal type.
0: Non-unit
1: Unit

descra(4) Array base.
0: C/C++ compatible Not Supported
1: Fortran compatible

descra(5) Repeated indices.
0: Unknown
1: No repeated indices

val( )	Array of length *pntr(m+1)-1* (see below for *pntr*) containing all the nonzero entries, and maybe some zero entries of *A*. *A* must be a square triangular matrix (*m=k*). *val( )* is row-oriented if *A* is a lower triangular matrix (*descra(2)=1*) and column oriented if *A* is an upper triangular matrix (*descra(2)=2*). All entries from the first nonzero entry through the diagonal entry of a row (column) are stored.
pntr( )	Array of length *m+1* (A lower triangular) or *k+1* (A upper triangular) such that *pntr(i)* and *pntr(i)+1-1,* respectively, point to the location in *val* of the first entry and last entry of the Skyline profile in row (column) *i*. In any case, the last entry is the diagonal entry.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Name SVBRMM/DVBRMM/CVBRMM/ZVBRMM
Variable block row matrix-matrix multiply

Purpose Variable block row matrix-matrix multiply. These subprograms compute the matrix-matrix product AB, where A is a m-by-k sparse matrix, and B is a k-by-n matrix with $m=mb\ x\ lb$ and $k=kb\ x\ lb$. Optionally, A may be replaced by A^T or A^*, where A^T or A^* is a k-by-m matrix, and B is a m-by-n matrix. Here A^T is the transpose and A^* is the conjugate-transpose of A. The product may be stored in the result matrix C or optionally may 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 form

$$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C \qquad C \leftarrow \alpha A^* B + \beta C$$

Usage VECLIB:

SUBROUTINE **SVBRMM**
INTEGER*4 **transa, mb, n, kb, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)**
REAL*4 **alpha, beta**
REAL*4 **val(*), b(ldb,*), c(ldc,*), work(*)**

CALL SVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **DVBRMM**
INTEGER*4 **transa, mb, n, kb, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)**
REAL*8 **alpha, beta**
REAL*8 **val(*), b(ldb,*), c(ldc,*), work(*)**

CALL DVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **CVBRMM**
INTEGER*4 **transa, mb, n, kb, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)**
COMPLEX*8 **alpha, beta**
COMPLEX*8 **val(*), b(ldb,*), c(ldc,*), work(*)**

CALL CVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZVBRMM
INTEGER*4 transa, mb, n, kb, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SVBRMM
INTEGER*8 transa, mb, n, kb, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DVBRMM
INTEGER*8 transa, mb, n, kb, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CVBRMM
INTEGER*8 transa, mb, n, kb, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZVBRMM
INTEGER*8 transa, mb, n, kb, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZVBRMM (transa, mb, n, kb, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

Input

transa	Indicates how to operate with the sparse matrix. 0: Operate with matrix 1: Operate with transpose matrix 2: Operate with conjugate-transpose matrix
mb	Number of block rows in matrix *A*.
n	Number of columns in matrix *C*.
kb	Number of block columns in matrix *A*.
alpha	Scalar parameter.
descra( )	Descriptor argument. Five element integer array.
descra(1)	Matrix structure. 0: General 1: Symmetric 2: Hermitian 3: Triangular 4: Skew (Anti)-Symmetric 5: Diagonal
descra(2)	Upper/Lower triangular indicator. 1: Lower 2: Upper
descra(3)	Main diagonal type. 0: Non-unit 1: Unit
descra(4)	Array base. 0: C/C++ compatible Not Supported 1: Fortran compatible
descra(5)	Repeated indices. 0: Unknown 1: No repeated indices
val( )	Scalar array of length *nnz* containing matrix entries.

indx(*)	Integer array of length *bnnz+1* such that the *i*-th element of *indx( )* points to the location in *val* of the (1,1) element of the *i*-th block entry.
bindx( )	Integer array of length *bnnz* consisting of the block row indices of the block entries of *A*.
rpntr( )	Integer array of length *mb+1* such that *rpntr(i)-rpntr(1)* is the row index of the first point row in the *i*-th block row. *rpntr(m;+1)* is set to *m+rpntr(1)*. Thus, the number of point rows in the *i*-th block row is *rpntr(i+1)-rpntr(i)*.
cpntr( )	Integer array of length *kb+1* such that *cpntr(j)-cpntr(1)* is the column index of the first point column in the *j*-th block column. *cpntr(kb+1)* is set to *k+cpntr(1)*. Thus, the number of point columns in the *j*-th block column is *cpntr(j+1)-cpntr(j)*.
bpntrb( )	Integer array of length *mb* such that *bpntrb(i)-bpntrb(1)* points to location in *bindx* of the first block entry of the *j*-th row of *A*.
bpntre( )	Integer array of length *mb* such that *bpntre(i)-bpntre(1)* points to location in *bindx* of the last block entry of the *j*-th block column of *A*.
lb	Dimension of dense blocks composing *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.
beta	Scalar parameter.
c( )	Rectangular arrary with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. Not used.
lwork	Length of *work* array.

Name SVBRSM/DVBRSM/CVBRSM/ZVBRSM
Variable block row format triangular solve

Purpose Variable block row format triangular solve. Given a scalar α, an upper- or lower-triangular sparse matrix A, and a m-by-n matrix B with m=mb x lb, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$, or $\alpha DA^{-1}B$, or $\alpha A^{-1}DB$, where D is a diagonal matrix. The size of A is m-by-m. Optionally, A^{-1} may be replaced by A^{-T}, or by A^{-*}. Here, A^{-T} is the transpose-inverse and A^{-*} is the conjugate-transpose-inverse of A. The solution matrix may be stored in the result matrix C or optionally may 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 solution matrix and the result matrix. Specifically, these subprograms compute matrix solutions of the form

$$C \leftarrow \alpha A^{-1}B + \beta C \qquad C \leftarrow \alpha A^{-T}B + \beta C \qquad C \leftarrow \alpha A^{-*}B + \beta C$$
$$C \leftarrow \alpha DA^{-1}B + \beta C \qquad C \leftarrow \alpha DA^{-T}B + \beta C \qquad C \leftarrow \alpha DA^{-*}B + \beta C$$
$$C \leftarrow \alpha A^{-1}DB + \beta C \qquad C \leftarrow \alpha A^{-T}DB + \beta C \qquad C \leftarrow \alpha A^{-*}DB + \beta C$$

Usage VECLIB:

SUBROUTINE **SVBRSM**
INTEGER*4 **transa, mb, n, unitd, blda, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)**
REAL*4 **alpha, beta**
REAL*4 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL SVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **DVBRSM**
INTEGER*4 **transa, mb, n, unitd, blda, ldb, ldc, lwork**
INTEGER*4 **descra(*), indx(*), rpntr(*), cpntr(*), bindx(*), bpntrb(*), bpntre(*)**
REAL*8 **alpha, beta**
REAL*8 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL DVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CVBRSM
INTEGER*4 transa, mb, n, unitd, blda, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE ZVBRSM
INTEGER*4 transa, mb, n, unitd, blda, ldb, ldc, lwork
INTEGER*4 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*16 alpha, beta
COMPLEX*16 val(*), b(ldb,*), c(ldc,*), work(*)
CALL ZVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

VECLIB8:

SUBROUTINE SVBRSM
INTEGER*8 transa, mb, n, unitd, blda, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
REAL*4 alpha, beta
REAL*4 val(*), b(ldb,*), c(ldc,*), work(*)
CALL SVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE DVBRSM
INTEGER*8 transa, mb, n, unitd, blda, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
REAL*8 alpha, beta
REAL*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL DVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE CVBRSM
INTEGER*8 transa, mb, n, unitd, blda, ldb, ldc, lwork
INTEGER*8 descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)
COMPLEX*8 alpha, beta
COMPLEX*8 val(*), b(ldb,*), c(ldc,*), work(*)
CALL CVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

SUBROUTINE **ZVBRSM**
INTEGER*8 **transa, mb, n, unitd, blda, ldb, ldc, lwork**
INTEGER*8 **descra(*), indx(*), bindx(*), rpntr(*), cpntr(*), bpntrb(*), bpntre(*)**
COMPLEX*16 **alpha, beta**
COMPLEX*16 **val(*), b(ldb,*), c(ldc,*), work(*)**
CALL ZVBRSM (transa, mb, n, unitd, dv, alpha, descra, val, indx, bindx, rpntr, cpntr, bpntrb, bpntre, b, ldb, beta, c, ldc, work, lwork)

Input

transa Indicates how to operate with the sparse matrix.
0: Operate with matrix
1: Operate with transpose matrix
2: Operate with conjugate-transpose matrix

mb Number of block rows in matrix *A*.

n Number of columns in matrix *C*.

unitd Type of scaling.

1. Identity matrix (argument *dv*( ) is ignored)
2. Scale on left (row scaling)
3. Scale on right (column scaling)

dv( ) Diagonal scaling array of length *lb*lb*mb*.

alpha Scalar parameter.

descra( )
Descriptor argument. Five element integer array.

descra(1) Matrix structure.
0: General
1: Symmetric
2: Hermitian
3: Triangular
4: Skew (Anti)-Symmetric
5: Diagonal

descra(2) Upper/Lower triangular indicator.
1: Lower
2: Upper

descra(3)	Main diagonal type. 0: Non-unit 1: Unit
descra(4)	Array base. 0: C/C++ compatible Not Supported 1: Fortran compatible
descra(5)	Repeated indices. 0: Unknown 1: No repeated indices
val( )	Scalar array of length *nnz* containing matrix entries.
indx( )	Integer array of length *bnnz+1* such that the *i*-th element of *indx( )* points to the location in *val* of the (1, 1) element of the *i*-th block entry.
bindx( )	Integer array of length *bnnz* consisting of the block column indices of the block entries of *A*.
rpntr( )	Integer array of length *mb+1* such that *rpntr(i)-rpntr(1)* is the row index of the first point row in the *i*-th block row. *rpntr(mb+1)* is set to *m+rpntr(1)*. Thus, the number of point rows in the *i*-th block row is *rpntr(i+1)-rpntr(i)*.
cpntr( )	Integer array of length *kb+1* such that *cpntr(j)-cpntr(1)* is the column index of the first point column in the *j*-th block column. *cpntr(kb+1)* is set to *k+cpntr(1)*. Thus the number of point columns in the *j*-th block column is *cpntr(j+1)-cpntr(j)*.
bpntrb( )	Integer array of length *mb* such that *bpntrb(i)-bpntrb(1)* points to location in *bindx* of the first block entry of the *j*-th block row of *A*.
bpntre( )	Integer array of length *mb* such that *bpntre(i)-bpntre(1)* points to location in *bindx* of the last block entry of the *j*-th block row of *A*.
b( )	Rectangular array with leading dimension *ldb*.
ldb	Leading dimension of *b*.

beta	Scalar parameter.
c( )	Rectangular array with leading dimension *ldc*.
ldc	Leading dimension of *c*.
work( )	Scratch array of length *lwork*. *lwork* should be at least *mb x lb x min(lb, n)*.
lwork	Length of *work* array.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z